Bachelorarbeit

Thema:

Konzeption und Implementierung eines NoSQL Laborversuches anhand einer MongoDB auf Basis eines Docker Containers

Projektbetreuer:
Prof. Dr. Volker Sänger

Zweitbetreuer:
M.Sc. Sai Manoj Marepalli

Hochschule Offenburg
Medien und Informationswesen

Michel Schwarz
Sommersemester 2019

Abstract

MongoDB – NoSQL für MI

Seit einigen Jahren erobern neue Datenbankmanagement-Systeme den Markt, die über das herkömmliche relationale Modell hinausgehen. Sie adressieren spezielle Anwendungsbereiche, in denen relationale Datenbanken nicht mehr genügen, z.B. bei der Verwendung riesiger Datenmengen oder unstrukturierter Daten. Aufgrund der zunehmenden Praxisrelevanz dieser Systeme wird es Zeit, dass sich M+I Studierende eingehend damit beschäftigen.

In der vorliegenden Arbeit soll deswegen ein Laborversuch konzipiert und realisiert werden, der das NoSQL-System MongoDB so adressiert, dass M+I Studierende den Umgang mit der MongoDB erlernen und dabei die wichtigen Eigenschaften des DBMS verstehen können.

Ziel ist es, dass Studenten die Eigenschaften der dokumentenorientierten Datenbank MongoDB erkennen und die wesentlichen Unterschiede zu relationalen Datenbanken ersichtlich werden.

Innerhalb des Laborversuches sollen die Studierenden mit einer grafischen Oberfläche zur Interaktion mit der MongoDB arbeiten. Hierzu werden ein Datenbankschema und eine exemplarische Anwendung entworfen und implementiert. Als Programmiersprache für die Interaktion ist Java mit den notwendigen Treibern für die MongoDB empfohlen worden.

Die Implementierung der MongoDB soll anhand des Containersystem Docker im Server integriert werden. Ziel des Containersystems ist es die Anwendung innerhalb des Laborversuches einfach adressieren und verwalten zu können. Auch können sich mit Hilfe des Containersystems zukünftige Projekte schneller und effizienter realisieren lassen.

Inhaltsverzeichnis

1. Einleitung 1
1.1. Eingrenzung der Bachelorarbeit 1
1.2. Aufbau der Arbeit 2
2. NoSQL Grundlagen 3
3. MongoDB 5
3.1. CAP-Theorem 6
3.2. Dokumente der MongoDB 7
3.3. Architektur 8
3.3.1. Storage Engine 9
3.3.2. Replica Sets 10
3.3.3. Oplog 11
3.3.4. Sharding 11
3.4. Verwendung von Indexen 12
3.5. Primärschlüssel 13
4. Installation mit Docker 15
4.1. Einsatzgebiete von Docker 15
4.2. Virtuelle Maschine vs. Docker 15
4.3. Docker Maschine 17
4.4. Docker Registry 17
4.5. Docker Network 18
4.6. Docker Images 19
4.6.1. Dockerfile 20
4.6.2. Docker-Compose 21
4.7. Docker im Datenbanken Labor 22
4.7.1. MongoDB Labor Image 22
4.7.2. Erstellen der MongoDB 23
4.7.3. Interaktion mit der MongoDB im Labor 24
5. MongoDB Inhalte im Laborversuch 27
5.1. Kollektionen und Dokumente 27
5.2. Dokumentorientiert mittels JSON 28
5.3. JSON Architektur 29
5.4. Datenschema validieren 31
5.5. Datenschema der MongoDB 32
5.6. Beziehungen zwischen Daten und Dokumenten 33
5.6.1. Embedded Documents 33
5.6.2. Document References 34
5.6.3. Gemischte Beziehungen 35
5.6.4. Zwei-Wege Referenzierung 35
5.7. Mediendaten 36
6. Adressierung der MongoDB mit Java 37
6.1. Die MongoDB Java Treiber 37

6.2. Verbindungsaufbau 38
6.2.1. Verbindung mit ConnectionString Klasse 38
6.2.2. Verbindung mit MongoClientSettings Klasse 39
6.3. BSON Dokumente 40
6.4. Dokumente mit Java anlegen 41
6.5. Dokumente in Java darstellen 42
6.6. Mongo CRUD 43
6.7. Atomare Operationen 48
6.8. Map-Reduce 49
6.9. Aggregation Pipeline 50
7. Durchführung des Laborversuches 51
7.1. Die Datensätze in der MongoDB 51
7.2. Die Java Beispielapplikation 53
7.3. Implementierung der Methoden 54
7.4. Beschreibung des Laborversuches 55
7.5. Verbindungsaufbau herstellen 55
7.6. Aufgabenbereich 1: Erstellung von JSON Dokumenten 56
7.7. Aufgabenbereich 1: Interpretation einer Datenstruktur 57
7.8. Aufgabenbereich 2: Adressierung der Daten mit Java Methoden 58
7.9. Aufgabenbereich 3: Adressierung eines Social Media Posts 69
8. Vorbereitung auf den Laborversuch 74
9. Zusammenfassung und Fazit 75
10. Literaturverzeichnis 77
11. Abbildungsverzeichnis 82
12. Anhang 85

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der ausgewiesenen Hilfsmittel angefertigt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach anderen gedruckten oder im Internet verfügbaren Werken entnommen sind, wurden von mir durch genaue Quellenangaben kenntlich gemacht.

Ort, Datum

_____

Michel Schwarz

Begriffsdefinitionen

API	Application Programming Interface
BSON	Binary JSON
CRUD	Create, Read, Update, Delete
DBMS	Datenbank Management System
GSON	Google Gson
GUI	Graphical User Interface
IP	Internet Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
MVC	Model-View-Controller
NoSQL	Not-Only SQL
OOP	Objektorientierte Programmierung
PDF	Portable Document Format
PHP	PHP: Hypertext Preprocessor
RDBMS	Relationales Datenbank Management System
REGEX	Regular Expression
SQL	Structure Query Language
TTY	TeleTYpewriter
VM	Virtuelle Maschine
XML	Extensible Markup Language
YAML	Yet Another Markup Language

1. Einleitung

In den letzten Jahren haben sich die Datenmengen im Internet immer weiter erhöht. Die Steigerung dieser Datenmengen hat viele Gründe. Es hat sich der Teil der Gesellschaft, der täglich das Internet nutzt, immer mehr zur größten Gruppe entwickelt. Daten werden somit durch die unterschiedlichsten Aktivitäten erzeugt. Wo früher noch Zeitungen gelesen wurden, werden heute Nachrichten über das Smartphone empfangen. Die Nutzung des mobilen Internets ist nicht mehr weg zu denken. Aber nicht nur diese Aktivitäten erzeugen vielerlei Daten, auch Unternehmen verarbeiten mit modernen Technologien eine große Anzahl an Informationen. Social Media Portale, Suchmaschinen und der Online Handel sind nur einige Beispiele.

Studenten der Fakultät Medien und Informationswesen der Hochschule Offenburg bekommen in Vorlesungen und Laboren Inhalte aus dem Bereich der Datenbanken vermittelt. Diese Inhalte sollten nun erweitert werden. Im Datenbankenlabor wird deshalb ein neuer Laborversuch integriert werden, der die NoSQL Datenbank MongoDB für den Laborunterricht bereitstellt.

Ziel dieser Arbeit ist es, dass die wichtigsten Merkmale der MongoDB ermittelt und umfassend erläutert werden. Dieses gesammelte Wissen muss so aufbereitet werden, dass die Laborteilnehmenden einen sehr umfangreichen Einblick in die Kernkompetenzen des DBMS bekommen. Dazu wurde ein Beispielprojekt mit Hilfe der aktuellen MongoDB Java Treiber implementiert, welches den Studierenden als Grundlage dient. Gleichzeitig soll die Erstellung der MongoDB und die notwendigen Konfigurationen so bereitgestellt werden, dass der administrative Aufwand sehr gering ist.

1.1. Eingrenzung der Bachelorarbeit

Die MongoDB ist eine Datenbank, die auf eine vielseitige Art und Weise eingesetzt werden kann. Üblicherweise wird in einer produktiven Umgebung eine MongoDB in einem Cluster aus mehreren Datenbanken betrieben. Das heißt, dass nicht nur eine einzelne Datenbank implementiert wird, sondern gleich mehrere. Innerhalb dieser Arbeit wurde kein Cluster erstellt, da dieses Prinzip nicht den Anforderungen im Labor entsprechen würde. Diese Arbeit hatte auch nicht zum Ziel, dass ein vollumfängliches Java Projekt erstellt wird. Das Beispielprojekt, mit dem die Studenten innerhalb des Labors interagieren, dient lediglich zur Visualisierung von Ergebnissen und soll einen besseren Einstieg in das Datenschema gewährleisten.

1.2. Aufbau der Arbeit

Der Laborversuch soll eine MongoDB Datenbank so adressieren, dass Studenten damit interagieren können. Dazu wird die Datenbank innerhalb des Containersystems Docker installiert. Das Containersystem wird dazu verwendet, die Datenbank flexibel zu adressieren, das bedeutet unabhängig von einer virtuellen Maschine oder Serverarchitektur. Es ermöglicht, die Implementierung mit einem Minimum an administrativem Aufwand zu starten. Dabei ist es wichtig, dass die spezifischen Einstellungen, die innerhalb des Labors benötigt werden, vordefiniert sind und nicht bei jedem Laborversuch neu angelegt werden müssen.

Um den Laborversuch konzipieren zu können, musste zuvor definiert werden, welche Inhalte vermittelt werden. Dabei ging es vor allem um die wichtigsten Unterschiede zwischen den bekannten RDBMS und NoSQL Systemen. Eine zentrale Entscheidung ist die Auswahl der geeigneten Datenbank. Hierbei wurde die MongoDB ausgewählt, da sie momentan das populärste und gefragteste DBMS ist. Anhand von Diagramm 1 ist zu sehen, dass die meisten Kenntnisse im Bereich der Datenbanken, die im Jahre 2018 weltweit gesucht wurden, auf die MongoDB zurückgingen. Folglich ist es nur von Vorteil, wenn im Datenbank Labor das meist gefragte System zum Einsatz kommt.

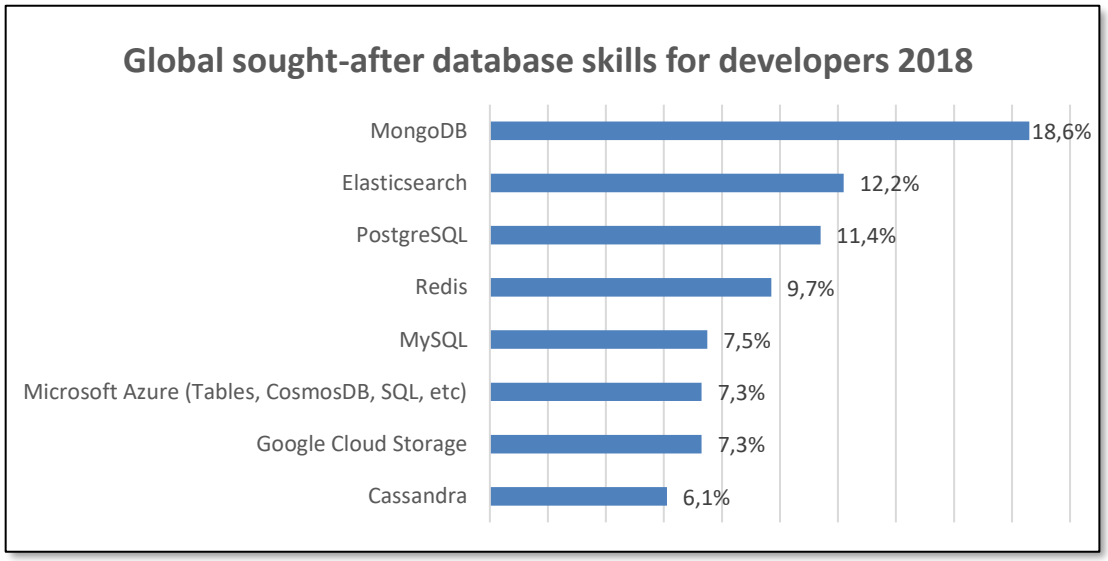


Diagramm 1: Die meistgesuchten Datenbank-Kenntnisse bei Softwareentwicklern weltweit ab Januar 2018 [1]

Um eine Manipulation der Daten zu ermöglichen, wurden die Java Treiber der MongoDB implementiert. Java ist eine der meist verwendeten Programmiersprachen. Da sie als Grundlage für OOP im Studium erlernt wird, sind die Studenten mit der Syntax vertraut. Alle relevanten Java Methoden werden ermittelt, um mit der MongoDB Daten auszutauschen. Die wichtigsten Elemente, die dabei beachtet werden sollten und wie eine Datenbankstruktur erstellt werden kann, sollen somit erlernt werden.

2. NoSQL Grundlagen

Damit die steigende Masse an Daten verarbeitet und gespeichert werden kann, war es notwendig, alternative Technologien im Bereich der Datenbanken zu entwickeln. Daraus sind das Leitbild und die Konzepte für die NoSQL Datenbanken entstanden. Diese sollten neue Möglichkeiten der Datenspeicherung bieten und zielgerichteter auf die Aufgabengebiete angepasst sein. Werden die NoSQL Datenbanksysteme einmal grob überblickt, wird ersichtlich, dass es eine große Auswahl an verschiedensten Systemen gibt. Dazu zählen z.B. Graphendatenbanken wie Neo4j, Key-Value Datenbanken wie Redis oder dokumentenorientierte Datenbanken wie die MongoDB. Aber was genau unterscheidet diese verschiedenen Systeme von den etablierten RDBMS? Um diese Frage zu beantworten, wird zunächst der Ansatz betrachtet, den die gängigsten NoSQL Datenbanken verfolgen.

Unter dem Begriff NoSQL sind viele unterschiedliche Systeme zu finden und alle haben ihre spezifischen Aufgabenbereiche. Dennoch verbindet die meisten ein ähnlicher Ansatz, der sie von den RDBMS unterscheidet. Sie folgen nicht alle den Richtlinien, denen die relationalen Systeme unterliegen. Dadurch entsteht eine deutlich größere Flexibilität und eine höhere Anpassungsfähigkeit an massive Datenströme und deren unstrukturierte Daten. NoSQL Technologien sind meist da im Einsatz, wo verteilte Systeme umfangreiche und flexible Daten erfassen sollen. Zudem werden Daten auch immer mehr in Echtzeit generiert. Um noch detaillierter widerspiegeln zu können, wie sich NoSQL Systeme von RDBMS unterscheiden, wird zunächst der relationale Ansatz erläutert. Dieser wird durch das ACID Prinzip beschrieben. Dieses Prinzip legt fest, wie Transaktionen in einer relationalen Datenbank vollzogen werden sollen. Dabei ist nicht die Prozedur in einem Programmcode gemeint, sondern vielmehr die Art wie Daten gespeichert werden. ACID ist durch die vier Begriffe ***Atomicity***, ***Consistency***, ***Isolation*** und ***Durable*** zusammengesetzt.

Die ***Atomarität*** beschreibt dabei die Tatsache, dass Daten nur eingetragen werden, wenn der Prozess vollständig abgeschlossen ist. Wird eine Transaktion nicht korrekt durchgeführt, müssen sämtliche Einträge wieder rückgängig gemacht werden.

Werden Transaktionen durchgeführt, müssen die Daten sich danach wieder in einem ***Konsistenten*** Zustand befinden. Das bedeutet, dass der Datenbestand keine Widersprüche aufweist.

Die Parallele Nutzung von Datensätzen erzeugt keine negativen Auswirkungen auf den Datenbestand. Die ***Isolation*** stellt dies sicher**.**

Dauerhaftigkeit besagt, dass alle getätigten Transaktionen dauerhaft in der Datenbank enthalten sein müssen. Systemausfälle dürfen den Datenbestand nicht beeinflussen [2, S. 13].

Datenbanken, die nach dem ACID Prinzip agieren, sind nicht geeignet für die Ansprüche an Skalierbarkeit und Verfügbarkeit. Diese sind für massive Daten aber notwendig. Einige NoSQL Datenbanken verstoßen deshalb bewusst gegen dieses Prinzip oder gegen Teile dieses Prinzips. Deshalb wurden die NoSQL Eigenschaften mit dem BASE Prinzip charakterisiert [2, S. 15]. Dieses beschreibt, wie NoSQL Datenbanken den Ansprüchen an Skalierbarkeit und Verfügbarkeit gerecht werden. Es besagt, dass die Performance erhöht wird, aber die Konsistenz sich bewusst in einem Übergangszustand befindet. Das BASE Prinzip steht für ***Basically Available***, ***Soft state*** und ***Eventual consistency***.

Basically Available beschreibt ein System, das grundsätzlich verfügbar ist, aber dass Teile diese Systems dennoch ausfallen können.

Soft state besagt, dass die Daten nicht wirklich konsistent sind. Der Zustand der Daten befindet sich in einem zeitlich variablen Zustand.

Eventual consistency bezieht sich auf einen längeren Zeitraum, in dem die Daten einen konsistenten Zustand erreichen könnten. Vorausgesetzt dass in dieser Phase keine weiteren Daten-Manipulationen vorgenommen werden [2, S. 16].

Die NoSQL Datenbanken haben aber noch weitere Merkmale, die erwähnt werden sollten. Zum einen sollte beachtet werden, dass durch die horizontale Verteilung der Daten mitunter Kosten reduziert werden können. Dies geschieht, da Systeme mit einem hohen Maß an Rechenleistung nicht erforderlich sind. Stattdessen sind mehrere kleinere Systeme ausreichend, die zusammen ein Cluster aus Datenbanken ergeben. Da die Mehrzahl aber als Open Source Projekte entstanden sind, besteht ein Mangel an Support. Zuverlässige Unterstützung kann infolgedessen nicht gewährleistet werden [2, S. 17-18]. Es entsteht auch eine Abhängigkeit zum jeweiligen System, da so ziemlich alle NoSQL Datenbanken kein SQL integriert haben. Die Sprache zur Datenmanipulation ist oftmals nicht konform mit anderen Systemen.

3. MongoDB

Die MongoDB wurde im Jahr 2009 von den Gründern Horowitz und Merriman als Open Source Datenbank bereitgestellt [2, S. 25]. Der Kerngedanke der Entwickler war, eine Datenbank zu entwerfen, die eine höhere Skalierbarkeit, Flexibilität und Geschwindigkeit hat. Das Prinzip sollte dennoch auf einer einfachen Handhabung beruhen. Wohingegen RDBMS mittels SQL arbeiten, trennt sich die MongoDB, wie viele andere NoSQL Datenbanken, von dieser Sprache zur Definition der Datenbank. Um massive Daten speichern zu können ist SQL nicht die passende Wahl. Es hat sich gezeigt, dass je größer die Menge an Daten ist und je mehr Tabellen in einer Anfrage enthalten sind, desto größer ist der Performanceverlust durch SQL [3, S. 297-298]. Es wurde ein anderes Konzept verfolgt, um die Daten zu definieren. Die MongoDB verwendet Dokumente und wird somit auch schon dem Kerngedanken der Flexibilität gerecht. Die Verwendung von Dokumenten führt zu einer strukturlosen Zusammensetzung an Daten, wo ein Dokument eine Vielzahl an unterschiedlichsten Informationen enthalten kann. Es ist nicht notwendig, mehrere Tabellen anzufragen, um einen einzelnen Datensatz zu erhalten. Dokumente können ein einzelnes Objekt repräsentieren, somit muss nur ein Dokument angefragt werden, um ein Objekt wiederzugeben. Dass die MongoDB keinem Schema folgt, bedeutet aber nicht, dass Architekturentscheidungen bei der Datenmodellierung ignoriert werden können. Es findet eine Verlagerung des Schema-Managements auf die Anwendungsentwicklung statt. Dies wiederum ist einer der vielen Gründe, warum die MongoDB eine solch hohe Popularität besitzt. Entwickler können in ihrem Programmcode die Struktur der Daten sehr flexibel gestalten und sind nicht von Integritätsbedingungen betroffen. Auch kann aufgrund der nicht integrierten SQL die Adressierung der Datenbank in ein und derselben Programmiersprache stattfinden. Es sollte aber beachtet werden, dass die Anwendungsentwickler damit mehr in den Gestaltungsprozess der Datenbank integriert werden [4, S. 431-433]. Die MongoDB ist bei Entwicklern auch dadurch sehr gefragt, da sie die populärsten Programmiersprachen unterstützt. Dazu zählen geläufige, wie z.B. Java, JavaScript oder PHP. Aber auch für Ruby oder Scala können Treiber integriert werden.

Die MongoDB war aus der Sicht der Entwickler nicht als eine Allround-Datenbank gedacht. Deshalb sollte vor einer Implementierung festgestellt werden, wozu das System entwickelt wurde. Die Verwendung der MongoDB ist darauf ausgelegt, in einem kurzen Zeitraum sehr viele Daten zu verarbeiten [5, S. 487-488]. Das liegt unter anderem an der Skalierbarkeit. In relationalen Systemen wird mit einer steigenden Anzahl an Datensätzen die Datenbank vertikal erweitert. Das heißt die Leistung des Systems wird erhöht, um mehr Daten verarbeiten zu können. Dies können mehr oder größere Tabellen sein. Dokumente hingegen werden in einer Kollektion gesammelt. Diese können als Äquivalent einer Tabelle angesehen werden, nur mit dem Unterschied, dass diese sehr einfach horizontal erweitert werden können. Steigende Datenmengen können somit auf mehreren Systemen verteilt werden, anstatt ein einzelnes leistungsfähiges System zu verwenden [6, 324–325].

3.1. CAP-Theorem

Die verschiedenen Systeme lassen sich, je nach dem für welche Aufgabenbereiche sie konzipiert wurden, unterschiedlich charakterisieren. Da keines der Systeme ein Alleskönner sein kann, muss klar zwischen dem Sinn und Zweck unterschieden werden. Bei verteilten Systemen wird das CAP-Theorem zu Hilfe genommen. Der Informatiker Eric Brewer von der Universität Berkeley, hat Anfang des Jahres 2000 die Annahme hergeleitet, dass ein System nicht alle drei Kerneigenschaften abdecken kann. Diese Kerneigenschaften sind die ***Consistency*** (Konsistenz), die ***Availability*** (Verfügbarkeit) und die ***Partition Tolerance*** (Ausfalltoleranz) [7, S. 33-34]. Wenn das CAP-Theorem in Bezug auf die MongoDB betrachtet wird, zeigt sich, dass diese sich auf Konsistenz und die Ausfalltoleranz fokussiert. Um dies besser verstehen zu können, betrachten wir die einzelnen Kerneigenschaften des Theorems etwas genauer.

Die ***Konsistenz*** **(C)** beschreibt den Zustand der Daten. Werden Daten in einem verteilten System eingefügt, muss der Zustand in jedem System gleich sein. Der Zustand der Daten soll somit im gesamten System übereinstimmen.

Ein System, das zu jedem Zeitpunkt erreichbar ist und eine kurze Reaktionszeit aufweist, besitzt eine hohe ***Verfügbarkeit*** **(A)**. Anfragen, die an die Datenbank gestellt werden, sollen in einem möglichst kurzen Zeitfenster beantwortet werden.

Verteilte Systeme bauen auf mehreren Bestandteilen auf. Sollte nun eines dieser Bestandteile ausfallen, darf das System in seiner Funktionsweise nicht beeinträchtigt werden. Es muss in der Lage sein, die Fehler zu umgehen, um weiterhin vollständig zu funktionieren. Die ***Ausfalltoleranz*** **(P)** steht für dieses Verhalten innerhalb eines Systems [2, S. 15].

Die MongoDB ist also durch die Definition der einzelnen Kerneigenschaften darauf ausgelegt, den Zustand der Daten in einem verteilten System konsistent zu halten und eine hohe Ausfalltoleranz zu gewährleisten. Das bedeutet aber nicht, dass die Verfügbarkeit bei der MongoDB keine Beachtung findet, sie muss sich lediglich den anderen Eigenschaften unterordnen.

3.2. Dokumente der MongoDB

Damit Daten ausgetauscht werden können, sind die Dokumente vom Datentyp JSON. Dieses Datenformat verwendet Schlüssel/Wert Paare, um die Daten abzubilden. Es bietet eine strukturlose Anordnung, was dazu führt, dass die Dokumente deutlich kompakter sind als bei anderen Formaten, wie z.B. XML. Die Strukturlosigkeit erhöht die Flexibilität, da Daten einfacher eingetragen und ausgelesen werden können. Jedes Dokument kann individuell aufgebaut sein [2, S. 26-27]. Ein weiterer Vorteil ist, dass die Dokumente ineinander geschachtelt werden können, um eine hierarchische Struktur zu erzeugen. Diese Einbettung in eine JSON-Datei wird als *"embedded document"* bezeichnet. Zusätzliche Informationen werden einem einzelnen Schlüssel zugeordnet, was die Struktur des Dokuments erheblich erweitert. Beziehungen zwischen Dokumenten müssen nicht mehr zwingend über Primär- oder Fremdschlüssel hergestellt werden, sondern die Daten befinden sich eingebettet unter einem Schlüssel im Dokument, auf den verwiesen werden kann. Dies erhöht die Performance bei Suchanfragen, da nicht wie im relationalen Modell mehrere Tabellen zusammengeführt werden müssen, um aufeinander referenzierende Daten zu erhalten [8, S. 37-38].

Das JSON Format wird verwendet, um Daten auszutauschen, aber nicht um die Daten in der Datenbank zu speichern. Hierfür wird das BSON Format verwendet, welches für *"Binary-JSON"* steht und eine gleiche Struktur wie das JSON Format besitzt. Es bietet aber mehr Möglichkeiten und Flexibilität bei der Datenspeicherung an. Dies kann als eine Erweiterung angesehen werden, die es Applikationen erleichtert, Daten zu verarbeiten. Die Zeit wird somit verringert, die zum Ausführen einer Anfrage benötigt wird. Auch können Programmiersprachen wie Java oder C# besser mit diesem Format umgehen. BSON Dokumente können aber durch die binäre Form mehr Speicherplatz brauchen, weil zum Abspeichern der Binärform mehr Daten benötigt werden. Auch muss der Aufwand betrieben werden, die beiden Dateiformate umzurechnen, wenn externe Daten ein- und ausgelesen werden. Aber es zeigt sich, dass aufgrund dieser Erweiterung die Performance erhöht wird. Dadurch ist der geringe Anstieg des Speicherplatzes vollkommen gerechtfertigt [8, S. 10-11]. Zu beachten ist auch, dass durch die Verwendung von Binärdaten, die Anzahl an Datentypen erhöht werden konnte. Durch das BSON Format können Datentypen wie beispielsweise *"ObjectID", "Integer"* (32Bit oder 64Bit)*, "Date", "Binary Data"* oder *"JavaScript"* Code eingefügt werden. Das ist eine deutliche Erweiterung gegenüber JSON [8, S. 36-37].

```
/* BSON: */
 \x16\x00\x00\x00               // total document size
  \x02                          // 0x02 = type String
  hello\x00                     // field name
  \x06\x00\x00\x00world\x00     // field value
\x00                            // End of object

/* JSON: */
{
 "hello":"world"                // "key":"value"
}
```

Abbildung 1: Schema des BSON und des JSON Formates [9, S. 319]

3.3. Architektur

Die MongoDB kann auf allen herkömmlichen Betriebssystemen implementiert werden und erzielt eine hohe Kompatibilität zu verschiedensten Anwendungen. Das liegt unter anderem daran, dass das System in C++ geschrieben wurde. Dadurch ist es weitgehendst unabhängig und kann ohne komplexe Anpassungen in Betrieb genommen werden. Zu beachten ist dabei lediglich, dass aus Performancegründen bei einem 32 Bit System die Datenbanken auf zwei Gigabyte Datengröße beschränkt werden [8, S. 7].

Wird die MongoDB auf einem Server oder einem lokalen System implementiert, werden drei Kernprozesse benötigt, damit die Datenbank in Betrieb genommen werden kann. Damit alle Operationen innerhalb des DBMS ausgeführt werden können, wird der *"mongod"* Prozess ausgeführt. Dieser bearbeitet eine Anfrage, welche an die Datenbank gestellt wird oder trägt Daten ein. Der *"mongod"* Prozess kann nur gestartet werden, wenn ein Dateiordner zum Ablegen der Daten erstellt wurde. Dieser muss auch dem Prozess in der Konfigurationsdatei der Datenbank zugeordnet werden. Besonders bei Linux Betriebssystemen ist zu beachten, dass die MongoDB auch die nötigen Schreib-Rechte über den Ordner besitzen muss. Über den Standard Port der MongoDB (Port 27017) kann nach erfolgreichem Start eine Verbindung hergestellt werden.

Der zweite wichtige Kernprozess startet die interaktive Mongo Shell. Der Prozess heißt *"mongo"* und wird allgemein dazu verwendet, um sich von administrativer Seite in die MongoDB einzuloggen. Alle Interaktionen mit der Datenbank können direkt in dieser Konsole vorgenommen werden.

Der Dritte Prozess ist *"mongos"* und ist ein Controller für verteilte Daten, die sich auf mehreren Datenbanken befinden. Dabei enthält jede MongoDB nur einen Teil des Datensatzes. Wird dieser aus einer Datenbank angefordert, ist es die Aufgabe dieses Prozesses, die Datenbank mit dem gesuchten Datensatz zu lokalisieren [2, S. 95-96]. Dieser ist aber nur notwendig, wenn auch ein verteiltes System aus mehreren Datenbanken konfiguriert worden ist. Solch ein verteiltes System wird als *"Replication Set"* bezeichnet. Dies findet oftmals Einsatz bei Cloud Systemen oder Applikationen, die eine hohe Verfügbarkeit oder Skalierbarkeit benötigen.

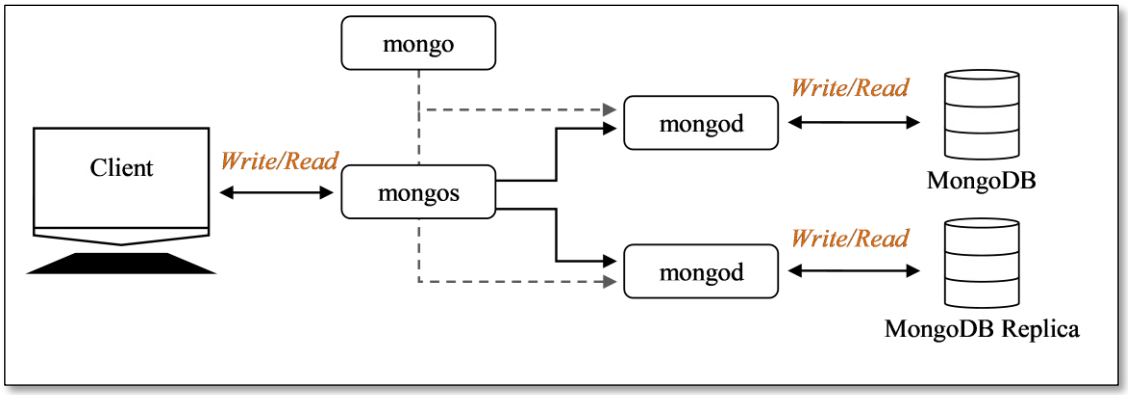


Abbildung 2: MongoDB Kernprozesse anhand von zwei vernetzten Datenbanken

3.3.1. Storage Engine

Wer anfängt, sich mit der MongoDB auseinander zu setzen, wird oft auf den Begriff *"Wired Tiger"* stoßen. Die Datenbank hat eine integrierte Engine, die das Speichern von Daten sowohl auf die Festplatte als auch den Hauptspeicher übernimmt. Die ursprüngliche Engine war *"MMAPv1"* (Memory-Map), welche vom Betriebssystem verwaltet wurde. Es wurden die wichtigsten Inhalte wie Indexe oder häufig gestellte Anfragen im Hauptspeicher gehalten. Dadurch ist die Leistung der MongoDB an den Hauptspeicher gebunden. Dabei konnte es passieren, dass die MongoDB allen zur Verfügung stehenden Hauptspeicher verwendet. Sollten die erforderlichen Daten zu groß sein oder andere Applikationen den Hauptspeicher belegen, kann diese Engine den Prozessen den Speicherplatz entziehen. Seit der Version 3.2 verwendet die MongoDB eine neue Storage Engine mit dem Namen *"Wired Tiger"*. Diese bewerkstelligt eine höhere Performance durch eine gleichzeitige Bearbeitung von Kollektionen. Des Weiteren komprimiert diese Engine die Daten, die abgespeichert werden mit unterschiedlichen Algorithmen. Je nach Datentyp können somit Daten um bis zu 70% komprimiert werden [2, S. 159-161]. Der Anwender der MongoDB kann selbst entscheiden, welche Engine verwendet werden soll. Die API bietet hierbei verschiedene Möglichkeiten der Konfiguration. Es sollte aber aufgrund der höheren Performance *"Wired Tiger"* als Standard Engine beibehalten werden. Dadurch wird die Verwaltung des Hauptspeicher optimiert, und ein stabileres System gewährleistet.

3.3.2. Replica Sets

Eine Applikation, die mit einer MongoDB verbunden ist, sollte zu jedem Zeitpunkt auf den Datenbestand zugreifen können. Laut des CAP-Theorems ist die MongoDB aber nicht direkt darauf ausgelegt eine hohe Verfügbarkeit zu bieten. Die Datenbank umgeht diesen Nachteil durch Replica Sets. Mehrere Datenbanken bilden ein Netzwerk, das den Datenbestand untereinander abgleicht und auf Basis einer primären Datenbank Redundanzen erzeugt. Dies gewährt zum einen die Sicherheit der Daten, zum anderen können Anfragen an den Datenbestand im Replica Set aufgeteilt werden. Die hierarchische Struktur wird dabei automatisch festgelegt. Sollte die primäre Datenbank ausfallen, übernimmt eine sekundäre dessen Position. Werden neue Daten eingetragen, kann dies nur von der primären Datenbank geschehen. Dies gewährleistet Konsistenz im Set, da nur eine Datenbank ihren Bestand ändert, alle anderen ihren angleichen. Sollte ein Datensatz von mehreren Manipulationen zur selben Zeit betroffen sein, kann nur die primäre Datenbank den Bestand ändern. Es tritt damit keine Diskrepanz im Datenbestand ein. Die MongoDB ist darauf ausgelegt, als Replica Set zu fungieren, das heißt um alle Vorteile des Systems nutzen zu können, sollte immer ein Replication Set angelegt werden [2, S. 97-99]. Um die Veränderungen im Datenbestand an die Replica Datenbanken zu senden wird ein *"Oplog"* (Operations log) verwendet.

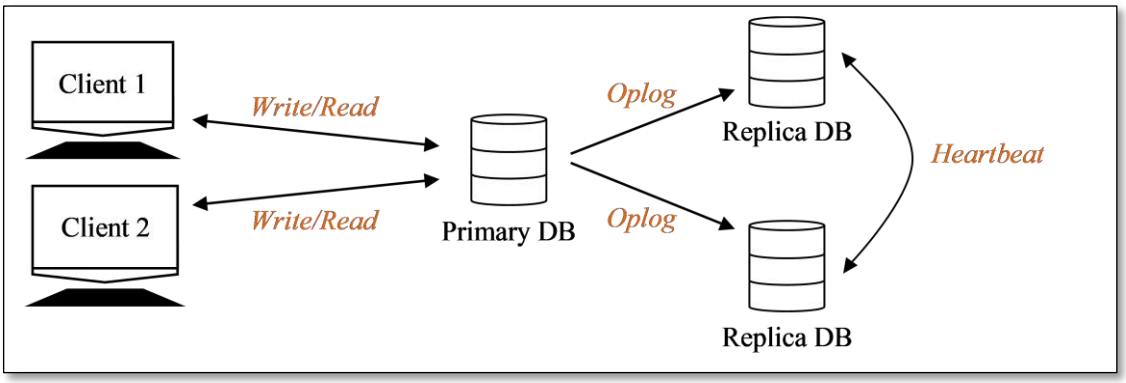


Abbildung 3: Visualisierung der Datenspeicherung in einem MongoDB Replica-Set

3.3.3. Oplog

Um Daten zwischen einer primären und einer sekundären Datenbank innerhalb eines Replication Sets auszutauschen, benötigt es eine Kollektion, die wie ein Logbuch angesehen werden kann. Diese Datei heißt *"Oplog"* und zeichnet alle Veränderungen in einer MongoDB auf. Es handelt sich hierbei um eine *"capped collection",* was bedeutet, dass diese eine definierte Größe hat. Diese wird in Megabyte angegeben. Sollte die maximale Größe der Kollektion erreicht werden, überschreiben die neuen Daten die ältesten. Das *"Oplog"* wird oftmals auch als Ringspeicher bezeichnet. Zwischen einer primären und sekundären Datenbank findet ein ständiger Abgleich dieser Kollektion statt. Das bedeutet, dass nicht das *"Oplog"* selbst von einer Datenbank zur anderen wechselt, sondern jede seine eigene Kopie besitzt.

Um nachzuvollziehen, wann neue Einträge hinzugefügt wurden, wird jeder Eintrag im *"Oplog"* mit einem Zeitstempel versehen. Dadurch ist es nur notwendig, die fehlenden Einträge zu übernehmen. Einer der wichtigsten Eigenschaften ist, dass bei einem Systemausfall, die Differenz des Datenbestandes ausgeglichen werden kann. Dabei ist aber zu beachten, dass je länger der Ausfall einer Datenbank ist, desto größer sollte das *"Oplog"* sein. Ist der Speicherplatz nicht ausreichend, können während des Systemausfalls eingetretene Änderungen nicht übernommen werden. Die Daten sind nicht länger konsistent [8, S. 289-290].

3.3.4. Sharding

Nachdem erläutert wurde, wie die MongoDB Konsistenz und Verfügbarkeit gewährleistet, sollte auch die horizontale Skalierbarkeit verdeutlicht werden. Dafür werden Datenbestände in Teilstücke zerlegt, diese werden als *"Shards"* bezeichnet. Diese *"Shards"* werden dann über zwei oder mehr Replica Sets verteilt. Wird der Datenbestand erhöht, können weitere Datenbanken hinzugefügt werden. Dadurch kann die Leistung der einzelnen Datenbanken erhöht werden, indem die Datenmenge der "*Shards*" geringgehalten wird. Da jede Datenbank lediglich für seinen eigenen Datenbestand verantwortlich ist, sind Suchanfragen schneller ausgeführt. *"Sharding"* ist besonders wichtig, wenn viele Daten gespeichert werden sollen. Eine MongoDB kann eine optimale Leistung erzielen, wenn die häufig angefragten Daten und meistverwendeten Indexe in den Hauptspeicher kopiert werden. Dies wird auch als das *"active work set"* bezeichnet. Übertrifft dieses *"work set"* die Größe des verfügbaren Hauptspeichers, sinkt die Leistung. Ist dies der Fall, kann über *"Sharding"* der Datenbestand aufgeteilt werden, ohne die Leistung zu beeinflussen [2, S. 124-126].

3.4. Verwendung von Indexen

Um eine hohe Performance zu erreichen, wird bei der MongoDB mit Indexen gearbeitet. Der Index führt Suchanfragen, die an die Datenbank gestellt werden, mit möglichst geringem Aufwand durch. Das bedeutet konkret, dass wenn eine Anfrage an die Datenbank gestellt wird, der Index dafür verantwortlich ist, dass nicht die gesamte Kollektion durchsucht wird. Die Datenbank verringert somit die Dokumente, die bei der Anfrage betrachtet werden müssen mit Hilfe der Indexe. Das Auslesen von Daten mittels einer Anfrage wird somit deutlich schneller durchgeführt. Dabei ist eine Möglichkeit, dass mit selbst definierten Indexen die Anfrage durchgeführt wird [8, S. 39-40].

Die MongoDB verwendet die Struktur eines Typ B-Baumes. Mit Hilfe dieser Indexstruktur kann die MongoDB den schnellstmöglichen Weg zur Anfrage herleiten. Die Herleitung des schnellsten Weges erfolgt bei einer erstmaligen Anfrage an das System. Dabei ermittelt die MongoDB über alle zur Verfügung stehenden Bearbeitungsmöglichkeiten den schnellsten Weg zum Ziel. Dieser allein wird dann vom System abgespeichert, um ihn für zukünftige Anfragen, die dieselben Kriterien erhalten, zu verwenden. Dabei ist aber zu beachten, dass bei dieser erstmaligen Anfrage die MongoDB deutlich mehr Zeit zum Ausführen braucht als bei allen folgenden Anfragen. Das liegt daran, dass bei der Generierung des Indexes die anderen Möglichkeiten der Anfragebearbeitung auch vollständig berechnet werden müssen [2, S. 78].

Die Generierung von Indexen hat aber nicht nur positive Effekte. Wird beispielsweise eine *"Write"* Operation durchgeführt, und dadurch der Inhalt von Feldern verändert, die mittels eines Indexes erfasst werden, kann sich das negativ auf die Performance auswirken. Der Grund dafür ist, dass die Indexe durch die Veränderung des Feldes von der Datenbank erneuert werden. Die *"Write"* Operation braucht durch die Erneuerung des Index dann mehr Zeit. Es kann also festgehalten werden, dass durch die Verwendung von Indexen die *"Read"*-Zeit deutlich verringert werden kann. Aber zugleich wird die benötigte Zeit für *"Write"* und auch *"Delete"* Operation erhöht [8, S. 39-40]. Die Indexe sollten daher auf die Abfragen möglichst genau angepasst sein und nicht zu allgemein gefasst werden. Auch sollten nur notwendige Indexe erstellt werden, um die Anzahl an zu bearbeitenden Indexen möglichst gering zu halten. Dies spielt eine wichtige Rolle bei der Erstellung eines Datenschemas und sollte vor allem bei einer Datenbank, die auf eine Vielzahl von Dokumenten ausgelegt ist, beachtet werden.

3.5. Primärschlüssel

Damit jedes Dokument in einer Kollektion eindeutig identifiziert werden kann, vergibt die MongoDB automatisch erzeugte *"_id"* Felder. Bei der automatischen Generierung wird der BSON Datentyp *"ObjectId"* eingefügt. Dieses Feld kann durch seine Einzigartigkeit als Primärschlüssel angesehen werden [2, S. 32]. Die Primärschlüssel werden erzeugt, wenn vom Datenbankdesigner hierfür kein Wert vorgegeben wird. Wenn die automatisch erzeugten ObjectIds mit dem Datenbankschema nicht vereinbar sind, kann stattdessen ein eigener Wert definiert werden. Das Feld *"_id"* muss dann beim Einfügen von Daten definiert werden. Dabei können verschiedene Datentypen, wie z.B. ein String verwendet werden. Dadurch ist die Erstellung der Primärschlüssel sehr flexibel gestaltet, allerdings bieten die von der MongoDB erstellten ObjectId´s einige Vorteile [10, S. 304].

Um die Vorteile der ObjectID besser verstehen zu können, betrachten wir zunächst dessen Aufbau. Das *"_id"* Feld besteht aus 24 Hex-Zahlen bzw. 12 Bytes. Die ersten vier Bytes sind ein Zeitstempel, drei Bytes dienen zur Identifizierung der Maschine, zwei weitere Bytes geben die Prozess-ID wieder und die letzten drei werden für einen zufälligen Zähler verwendet. Eine von der MongoDB automatisch erstellte ID sieht dann wie folgt aus: *"ObjectId("507f191e810c19729de860ea")"* [8, S. 38-39].

Die Vorteile sind, dass anhand dieses Aufbaus, zusätzliche Informationen aus der ID ausgelesen werden. Die Datenbank selbst kann über bestimmte Funktionen z.B. den Zeitpunkt der Erstellung ausgeben. Aber auch bei der Anwendungsentwicklung kann ein Java oder PHP Treiber zusätzliche Informationen aus der ObjectID ermitteln. Nachfolgend ist ein Java Beispiel für ein Dokument, das nur anhand der ObjectID den Zeitpunkt der Erstellung wiedergibt.

```
MongoCollection<Document> coll =
             MongoConnection.database.getCollection("name");

BasicDBObject query = new BasicDBObject( "vorname", "Peter" );
MongoCursor<Document> cursor = coll.find(query).iterator();

while (cursor.hasNext()) {
  System.out.print(cursor.next().getObjectId("_id").getDate().toString());
}
```

Abbildung 4: Beispiel-Code für das Auslesen des Datums aus einer ObjectID in Java

Das Ergebnis des Beispiel-Codes aus Abbildung 4 sehe dann wie folgt aus:

```
Sat Mar 30 17:23:41 CET 2019
```

Abbildung 5: Beispiel-Ausgabe in Java anhand des Datums einer ObjectID

4. Installation mit Docker

Das Containersystem Docker, das im Jahr 2013 erschienen ist, bietet einen optimierten Ansatz in der Virtualisierung auf Betriebssystemebene. Mit Hilfe von Docker können Entwicklungsumgebungen leichter realisiert und verwaltet werden. Die MongoDB im Datenbank Labor, soll innerhalb eines Docker Containers mittels eines Docker-Compose Files adressiert werden.

4.1. Einsatzgebiete von Docker

Diese Technologie wird wegen seiner vielseitigen und flexiblen Einsatzfähigkeit immer mehr verwendet. Docker kann ohne aufwändige Anpassungen in Linux, Windows oder Mac Betriebssystemen integriert werden. Administratoren können damit eine höhere Kompatibilität zwischen Systemen und Anwendungen erzeugen.

In klassischen Entwicklungsumgebungen war die Versionierung von Software und der Transport von Systemen meist mit einem hohen Aufwand verbunden. Treiber mussten auf eine einheitliche Version aktualisiert und fehlende Inhalte ergänzt werden. Docker bietet nicht nur die Möglichkeit an, diese Entwicklungsumgebungen in einzelne Container aufzuteilen, sondern bietet hierbei auch eine sehr einfache Adressierung an. Komplexe Architekturen können somit unabhängig vom System, der Hardware und notwendiger Software aufgebaut werden. Docker bietet zugleich einen Überblick in Form von Ebenen an. Das bedeutet konkret, dass wenn ein Container erstellt oder modifiziert wird, wird dies als neue Ebene hinzugefügt. Die Ebenen können als Verlauf des Containers angesehen werden. Einfache Rollbacks auf eine frühere Version sind somit problemlos möglich, indem auf eine ältere Ebene zurückgegangen wird [11, S. 2-3].

4.2. Virtuelle Maschine vs. Docker

Bevor Docker entwickelt wurde, war die Erstellung von Containern bereits bekannt. Beispielsweise durch Linux Container. Diese waren aber nicht sehr kompatibel mit anderen Systemen. Deshalb wurde oftmals eine virtuelle Maschine (VM) verwendet, die den Server unterteilt. Diese VMs sind noch immer eine häufig verwendete Methode zur Separierung von Infrastrukturen. Docker bietet hierbei ein ähnliches Konzept an, wobei beide Vor- und Nachteile haben. Beispielsweise können auf einem Server mehrere einzelne VMs eingesetzt werden, um mehrere kleinere Systeme zu virtualisieren. Diese einzelnen VMs benötigen aber viele Ressourcen, genauer gesagt brauchen alle ein vollumfängliches Betriebssystem mit BIOS und Speicherplatz [11, S. 6]. Vereinfacht kann gesagt werden, dass die VMs eine Zwischenschicht bilden, die dem Benutzer gezeigt wird und zwischen ihm und dem eigentlichen Betriebssystem sitzt.

Diese Schicht zu illusionieren ist aber im Vergleich zu Docker aufwändiger, was die Ressourcen angeht. Dagegen muss aber festgehalten werden, dass eine VM sich besser eignet, um direkt mit der Hardware zu interagieren. Sie kann spezifischer an die einzelnen Komponenten angepasst werden. Docker hingegen setzt auf eine höhere Dialogfähigkeit zwischen seinen Containern und dem tatsächlich vorhandenen Betriebssystem. Es verwendet einen definierten Bereich des Kernels, was dazu führt, dass die Soft- und Hardware zwischen allen laufenden Containern und den Betriebssystemen aufgeteilt wird. [12, S. 142].

Die Docker Container können dabei als eine isolierte Instanz betrachtet werden. Die Prozesse, die innerhalb eines Containers laufen, seien es nun einzelne oder mehrere, sind von denen in anderen Containern getrennt. Das gleiche gilt für die Prozesse des Betriebssystems, auf dem Docker installiert ist [13, S. 1]. Container sind auch sicherer in ihrer Handhabung und bieten eine höhere Skalierbarkeit. Es können durch den geringeren Bedarf an Ressourcen auf einem System deutlich mehr Container laufen als VMs. Einer der wichtigsten Faktoren ist aber der einfache Transport. Die Container können im System, in dem sie erstellt worden sind, exportiert und nahtlos in jedes andere beliebige System integriert werden [12, S. 146].

Festzuhalten ist somit, dass innerhalb der einzelnen Container von Docker nur wenige Datenmengen benötigt werden, um diesen mit einem integrierten Betriebssystem zu adressieren. Es ist dem Entwickler überlassen, ob er dieses weiter ausbaut oder in wie weit dieses modifiziert wird. Für vielerlei Anwendungen innerhalb eines Containers ist mitunter auch kein Betriebssystem notwendig, nur die Applikation selbst wird dann betrieben. Diese Tatsache führt dazu, dass nur ein geringer Anteil an Ressourcen (Bsp. Speicherplatz, Ram) benötigt wird. In der nachfolgenden Abbildung 6 ist der Unterschied noch einmal veranschaulicht.

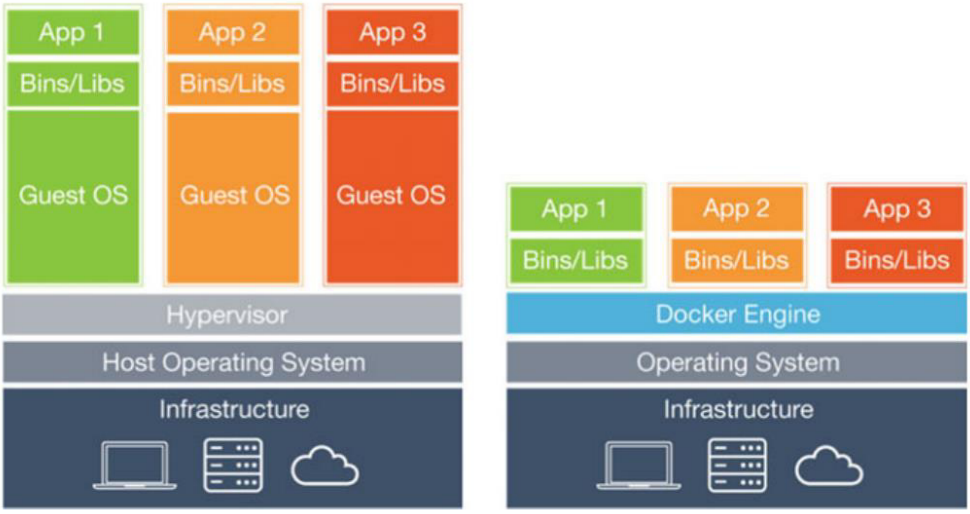


Abbildung 6: Virtuelle Maschine vs. Docker [12, S. 147]

4.3. Docker Maschine

Die Funktionsweise von Docker teilt sich auf mehrere Komponenten auf. Die erste ist der Docker-Client. Durch den Docker Client findet die Interaktion des Benutzers mit dem Docker-Daemon statt. Durch eine Programmierschnittstelle übergibt der Client die Befehle an den Daemon, welcher diese dann ausführt. Der Daemon befindet sich vor Empfang der Befehle in einem *"Listen"* Modus und lauscht auf die Eingaben des Benutzers [14, S. 32-33]. Beide Komponenten sind direkt auf dem Host-System installiert und werden auch als die Docker-Engine bezeichnet [15, S. 71]. Die vom Daemon erzeugten Ergebnisse, die nach der Durchführung eines Befehls entstehen, werden auch als Objekte bezeichnet. Dies kann exemplarisch ein Container sein, aber auch eine neue Version eines Container Images ist denkbar. Die Objekte können aber nicht ohne einen gegebenen Input generiert werden. Deshalb kommt eine weitere Komponente hinzu und diese ist die Docker Registry. Mit einer Verbindung zu einer Registry kann ein Docker-Daemon die Basis-Images für die Erstellung von Containern downloaden. Dabei muss zwischen öffentlichen Registrys, wie der cloudbasierten von Docker und einer privat genutzten unterschieden werden. Die Docker Registry ist somit eine Komponente, von der aus der Docker-Daemon die Informationen erhält, um Container zu erstellen [16, S. 122-123].

4.4. Docker Registry

Eine private Registry kann die Images beinhalten, die in einer Entwicklungsumgebung dokumentiert und aufbewahrt werden sollen. Die öffentliche ist die zentrale Stelle, um die Software zu erhalten, die in einem Container verwendet werden soll. In der offiziellen Registry sind Images für diverse Anwendungen zu finden, wie z.B. Betriebssysteme oder serverseitige Anwendungen. Werden Images modifiziert und umfangreich an die eigene Entwicklungsumgebung angepasst, dann können diese in einer eigenen Registry abgelegt werden. Dafür muss die Applikation Docker Registry implementiert werden. Werden neue Container vom Anwender erstellt, wird vom Daemon geprüft, ob sich in der privaten Registry ein Image befindet [14, S. 33]. Es ist somit möglich, individuell angepasste Images einfach und schnell abzulegen, aber vor allem sind diese dann nur in einem geschlossenen System erreichbar. In Abbildung 7 wird mittels des Befehls *"docker pull"* das Image aus der offiziellen Registry gedownloadet.

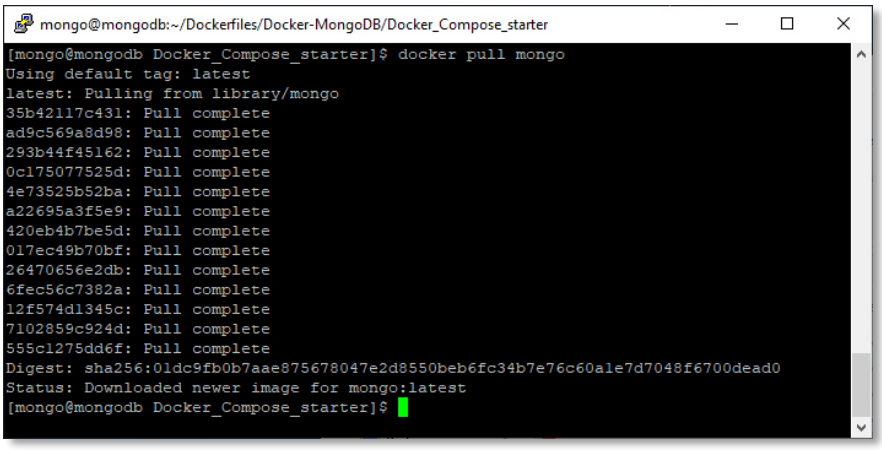


Abbildung 7: Download des offiziellen MongoDB Images

4.5. Docker Network

Der Datenaustausch und die Kommunikation zwischen einzelnen Anwendungen kann durch Docker Networks ermöglicht werden. Diese können so konfiguriert werden, dass alle Container, die zu einer Anwendung gehören, sich in einem eigenen Netzwerk befinden. Auch wenn vom Anwender nicht explizit ein eigenes Netzwerk konfiguriert wird, ist es von Vorteil, wenn verstanden wurde, wie der Docker Daemon mit den Containern interagiert.

Verschiedene Treiber ermöglichen dabei unterschiedliche Adressierungen. Beispielsweise kann ein Container keine Verbindung zu anderen Hosts, Containern oder sonstigen Anwendungen besitzen. Dies wird als *"None Networking"* bezeichnet. Es besteht aber auch die Möglichkeit, mittels des *"Overlay Networks",* eine Verbindung zu mehreren Docker Host herzustellen [11, S. 121-122]. Diese werden aber meist nur dann verwendet, wenn spezielle oder sehr weitreichende Anwendungen erstellt werden sollen.

Der wichtigste Treiber, mit dem jeder Container kommuniziert, ist *"Bridge"*. Dieser Treiber wird standardmäßig verwendet, wenn kein Netzwerk angegeben wird. Soll explizit ein neues Netzwerk neben dem Standard Treiber erstellt werden, so entsteht eine zusätzliche Isolierung. Container innerhalb des zusätzlichen *"Bridge"* Netzwerkes können die Ports untereinander adressieren. Außenstehende Anwendungen, sehen nur Ports die explizit veröffentlicht werden.

Anhand der MongoDB kann beispielhaft erläutert werden, wie der Standard Treiber *"Bridge"* die Kommunikation zum Container ermöglicht. Diese Datenbank soll durch externe Treiber, wie z.B. die offiziellen Java Treiber, adressiert werden können. Das bedeutet, dass die MongoDB einen Port und eine IP-Adresse benötigt. Wird nun ein Container erstellt, dann muss mittels eines *"-p"* Flags der Port veröffentlich werden.

```
docker run -dit --name mongodb -p 27017:27017 mongodb:data /bin/bash
```

Abbildung 8: Docker-Container wird mit Port gestartet

Die IP Adresse wird innerhalb des Standard Netzwerkes automatisch vergeben. Der Docker-Daemon leitet alle Anfragen, die beim Docker-Host auf den veröffentlichen Port eingehen, an den Container weiter. Alle anderen Container, die sich im Standard Netzwerk befinden, können ebenfalls mit der MongoDB interagieren. Sollte ein bestimmter Container nicht auf die MongoDB zugreifen dürfen, dann ist es sinnvoll, durch ein manuell erstelltes Netzwerk die Anwendungen zu isolieren [17].

4.6. Docker Images

Soll eine Anwendung aus einer beliebigen Registry in einem Container verwendet werden, dann wird zum Erstellen ein Image benötigt. Dieses besteht aus einem Namen, einer ID und einer Erkennungsmarke (tag). Werden Images einmal generiert, können sie nicht mehr verändert werden. Sie gelten als einzigartig und ihr Inhalt kann nur noch als Container wiedergegeben werden. Allerdings können Images schrittweise aufeinander aufbauen.

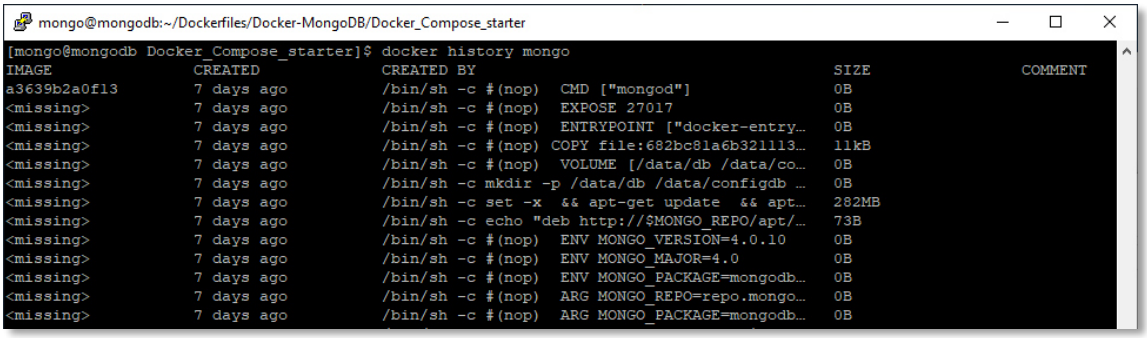


Abbildung 9: Docker History des offiziellen MongoDB Image

Wird der Aufbau eines Images durch den *"history"* Befehl betrachtet, dann ist ersichtlich, dass dieses in mehreren Ebenen dargestellt wird. Jeder Container, der durch ein Image generiert wird, kann modifiziert und beliebig verändert werden. Wird aus diesem modifizierten Container ein neues Image generiert, dann wird zum ursprünglichen Image eine neue Ebene hinzugefügt [14, S. 54-55]. Docker speichert somit jede Veränderung neu ab. In Bezug auf die Dateigröße eines Images bedeutet das, dass die älteren Versionen nach einer Modifikation noch vorhanden sind und die Dateigröße wächst.

In Abbildung 9 sind die verschiedenen Ebenen eines Images zu sehen. Der Hinweis *"<missing>"* besagt, dass auf die vorherige Version nicht zugegriffen werden kann. Grund hierführt ist, dass der Ursprung dieses Images in einem anderen System war. Die Informationen zum Erstellen dieser Ebene fehlen, die Daten sind aber dennoch vorhanden [18]. Um den benötigten Speicherplatz eines Images zu verringern, ist es möglich, die unteren Ebenen auf das Nötigste zu reduzieren. Dazu muss ein Container anhand dieses Images erstellt werden. Der erstellte Container kann dann durch einen *"export"* Befehl als verpackte Datei (.tar) gespeichert werden. Dabei werden nur die relevanten Ebenen beibehalten und zu einer zusammengefasst. Beim *"import"* Befehl wird aus der verpackten Datei wieder ein Image, welches nur noch diese eine Ebene beinhaltet [16, S. 191-192]. Dies sorgt für mehr Übersicht in der History, spart Speicherplatz und das Image lässt sich leichter transportieren.

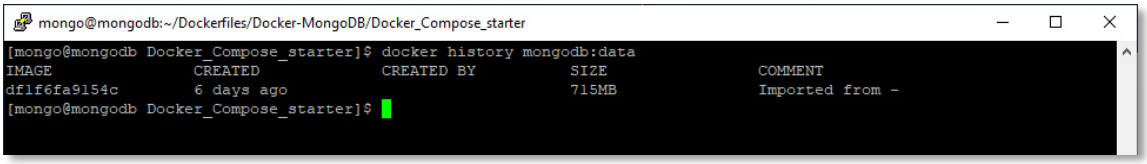


Abbildung 10: Die History des importierten MongoDB Containers

4.6.1. Dockerfile

Das Dockerfile ist eines der wesentlichen Bestandteile bei der Generierung von eigenen Images. Mit dieser Datei kann ein Ablauf definiert werden, der zur Erstellung eines neuen Images benötigt wird. In der ersten Zeile wird angegeben, auf welches ursprüngliche Image sich dieser Ablauf bezieht. Ohne diese Angabe kann kein Image erstellt werden. Docker bietet ein *"Scratch"* Image an. Anwendungen, die von Grund auf neu erstellt werden sollen, können auf diesem Image aufgebaut werden. Dieses ist die Basis für alle erstellten Images aus der Registry [16, S. 231]. In Abbildung 11 ist ein einfacher Aufbau eines Dockerfiles anhand eines Scratch Images zu sehen:

```
FROM scratch
COPY ./new_folder ~/
CMD ["echo","Image created"]
```

Abbildung 11: Beispiel für ein Dockerfile

Innerhalb eines Dockerfiles können Konsolenbefehle ausgeführt werden. Dateien können kopiert oder Ordner erstellt werden. Es ist somit möglich, den Aufbau einer Applikation sehr genau zu definieren. Zu beachten ist aber, dass jeder der einzelnen Befehle im Dockerfile wieder eine zusätzliche Ebene in das Image einfügt. Das Resultat eines Dockerfiles ist immer ein neues Image [16, S. 229].

```
docker build -t my-container:v1 ~/Home/
```

Abbildung 12: Build Befehl zum Erstellen eines Images aus einem Dockerfile

Der Befehl *"docker build"* sucht das Dockerfile im angegeben Ordnerpfad. Docker erkennt die Datei selbständig und beginnt damit das Image aufzubauen. Die *"-t"* Option weist dem neu erzeugten Image eine Erkennungsmarke im Format *"name:tag"* zu. Diese sollten immer angegeben werden, da sie die Übersicht über alle Images und daraus resultierenden Containern erheblich verbessern. In Abbildung 13 ist visualisiert, wie der Befehl *"docker build"* das Image erzeugt und aus einem Image mit Hilfe des *"docker run"* Befehls ein Container erstellt werden kann.

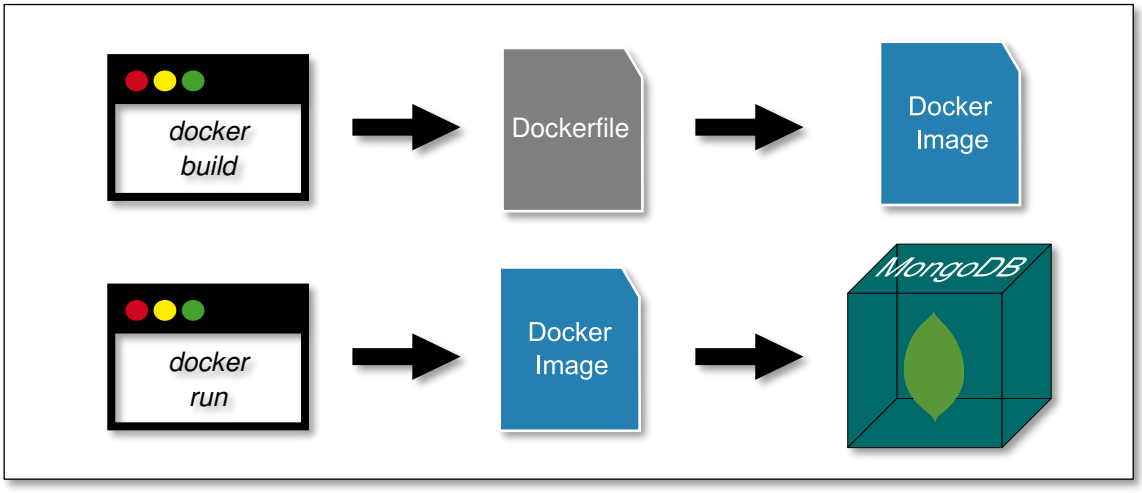


Abbildung 13: Erstellen eines Images und eines Containers

4.6.2. Docker-Compose

Mit Docker können eine Vielzahl an Containern anhand von verschiedenen Images gleichzeitig in Betrieb genommen werden. Mit zunehmender Anzahl an Containern steigt aber der administrative Aufwand. Wenn einzelne Container dann noch über ein Netzwerk miteinander interagieren sollen, ist die Übersicht schnell verloren. Auch sind Container oftmals abhängig voneinander, z.B. wenn eine Datenbank mit einem Server interagieren soll. Die Anwendungen werden sehr schnell sehr komplex und die verschiedenen Einstellungen der Container können nicht mehr überblickt werden. Bei einer großen Anzahl an Containern oder einer komplexen Struktur sollte deshalb nicht jeder einzelne Container manuell verwaltet werden.

Hierbei kann das Vorgehen erleichtert werden, indem Docker-Compose integriert wird. Docker-Compose ermöglicht es, in einer einzelnen Datei, die Konfiguration für eine Vielzahl von Containern zu definieren. Die Auszeichnungssprache YAML wird verwendet, um den Ablauf festzulegen. Dieser wird in der Datei, die immer die Bezeichnung *"docker-compose.yml"* haben muss, abgespeichert. Innerhalb der Beschreibung müssen die zu verwendenden Images angegeben werden. Docker-Compose ist so flexibel, dass es auch möglich ist, ein Image direkt über ein Dockerfile erstellen zu lassen. Äußere und innere Ports können mittels eines Mappings definiert werden. Dateien können durch die Angabe der Pfade vom Host in den Container importiert werden. Dies alles kann für mehrere Container in einer einzelnen Datei definiert werden. Docker-Compose kann dann anhand dieser Beschreibung die einzelnen Services mit ihren Netzwerken und Abhängigkeiten starten. Hierbei muss aber beachtet werden, dass Docker-Compose nicht nur die einzelnen Container definieren kann, sondern auch die Beziehungen untereinander [11, S. 151-152].

Um ein Compose-File zu starten, wird lediglich der Befehl *"docker-compose up"* benötigt. Im Gegensatz zu *"docker run"* müssen keine zusätzlichen Optionen angegeben werden. Das heißt keine Ports, Namen oder Netzwerke. Alle Informationen sind im Compose-File enthalten [11, S. 156-159].

Im Datenbankenlabor soll ein Compose-File verwendet werden, um die MongoDB mit allen notwendigen Daten zu starten und wieder zu entfernen. Hierbei wurden innerhalb des Compose-Files nicht nur eine, sondern zwei Datenbanken angelegt. Die zweite Datenbank dient lediglich als Kopie, um bei einem schwerwiegenden Implementierungsfehler während des Laborversuches auf eine bereits laufende und adressierbare Datenbank ausweichen zu können.

4.7. Docker im Datenbanken Labor

4.7.1. MongoDB Labor Image

Innerhalb des Datenbank Labors wurden Applikationen bislang in einer VM implementiert und gestartet. Für den Laborversuch mit der MongoDB sollte Docker verwendet werden, um eine höhere Flexibilität und Anpassungsfähigkeit zu erreichen. Dies gewährleistet, dass die MongoDB leicht auf verschiedenen Servern implementiert werden kann, sollte es einmal notwendig sein, die Applikation woanders einzusetzen. Der administrative Aufwand wird deutlich verringert, da der Container lediglich über ein Compose-File gestartet werden muss.

Ein bereits implementiertes Beispielprojekt, soll in der MongoDB enthalten sein und adressiert werden können. Im Laborversuch werden die Studierenden Daten aus der MongoDB anfordern und eintragen. Ein Image musste erstellt werden, welches alle hierfür notwendigen Daten und Einstellungen bereits enthält:

- Alle Datenbanken für die Laborgruppen (36x).
- Das Beispielprojekt in jeder Datenbank integriert.
- Alle Benutzer und deren Rechte definiert.
- Ein Administrator für alle Datenbanken.
- Der Pfad für die Speicherung der Daten muss im Container angelegt sein.

Der letzte Punkt musste durchgeführt werden, da bei dem offiziellen MongoDB Image die Dateien außerhalb des Containers gespeichert werden. Das liegt daran, dass die Daten beständig sein sollen. Sollte ein Container gelöscht werden, dann sind die Daten immer noch auf dem Hostsystem enthalten. Diese Konfiguration wurde abgeändert, so dass die MongoDB die Daten innerhalb des Containers ablegt. Wird der Container entfernt, sind die Daten der Laborteilnehmenden nicht mehr erhalten. Sobald der Container wieder erstellt wird, ist die Datenbank mit allen im Image enthaltenen Definitionen bereit. Sollten dennoch Modifikationen notwendig sein, kann auf ein zusätzlich bereitgestelltes Basis Image, welches nur die Datenbanken und die Benutzer beinhaltet, zurückgegriffen werden.

4.7.2. Erstellen der MongoDB

Das Compose File für den Start der MongoDB im Labor soll hier genauer betrachtet werden. Das Image, welches in das Docker-Compose File eingetragen wurde, ist eine modifizierte Version des offiziellen MongoDB Images. Wird das Compose-File ausgeführt, beginnt der Docker-Client damit, die einzelnen Container aufzubauen. Die Versionsnummer *"version: '3.5'"* gibt dabei an, welche Schlüsselwerte in der Datei verwendet werden [11, S. 154-156]. Wird vom Benutzer keine Version vorgegeben, dann wird Docker immer die Version 1 verwenden [15, S. 181]. In Abbildung 14 ist der Inhalt für das Ausführen des Compose-Files für die MonogDB zu sehen.

```
version: '3.5'

services:

  mongo:
    image: mongodb:data
    tty: true
    ports:
     - 27017:27017
    container_name: mongodb1
    hostname: mongo-container1
    entrypoint: ./usr/local/bin/docker-entrypoint.sh
    command: mongod --auth
```

Abbildung 14: Docker-Compose File für die MongoDB im Labor

Die einzelnen Konfigurationen haben folgende Funktionen [19]:

- Image: Das zu verwendende Image oder Dockerfile
- TTY: Aktivieren einer Pseudo Bash-Shell
- Ports 27017:27017: Legt ein Port Mapping vom Hostsystem zum Container fest.
- Container-Name: Festlegen des Namens des Containers
- Hostname: Hostname innerhalb des Containers
- Entrypoint: Festlegen, an welche Datei die Befehle bei Start gesendet werden. In diesem Fall ist *"docker-entrypoint.sh"* ein Skript, das die MongoDB Parameter überprüft. Diese stammt aus dem offiziellen Image der MongoDB und wurde übernommen.
- Command: Die Befehle, die beim Starten eines Containers ausgeführt werden. Hierbei handelt es sich um den Mongo Server im Authentifikationsmodus.

Wird zu Beginn eines Laborversuches die Datenbank mit dem Compose-File aufgebaut, kann direkt mit einer Applikation oder einem Java-Treiber auf die Datenbank zugegriffen werden. Alle notwendigen Modifizierungen sind bereites getätigt worden. Beim Abbau wird alles wieder nahtlos entfernt, das heißt es ist bereits alles für den nächsten Versuch bereit.

4.7.3. Interaktion mit der MongoDB im Labor

Innerhalb des Laborversuches wird *"Robo 3T"* zur Interaktion mit der Datenbank verwendet. Dies ist ein Grafisches User Interface, welches plattformunabhängig verwendet werden kann. Es bietet eine überschaubare Ansicht aller Kollektionen und derer Dokumente. Besonders von Vorteil ist, dass eine Mongo Shell integriert worden ist. Alle Methoden, die nicht von dieser Anwendung abgedeckt werden, können durch diese Shell ergänzt werden [20].

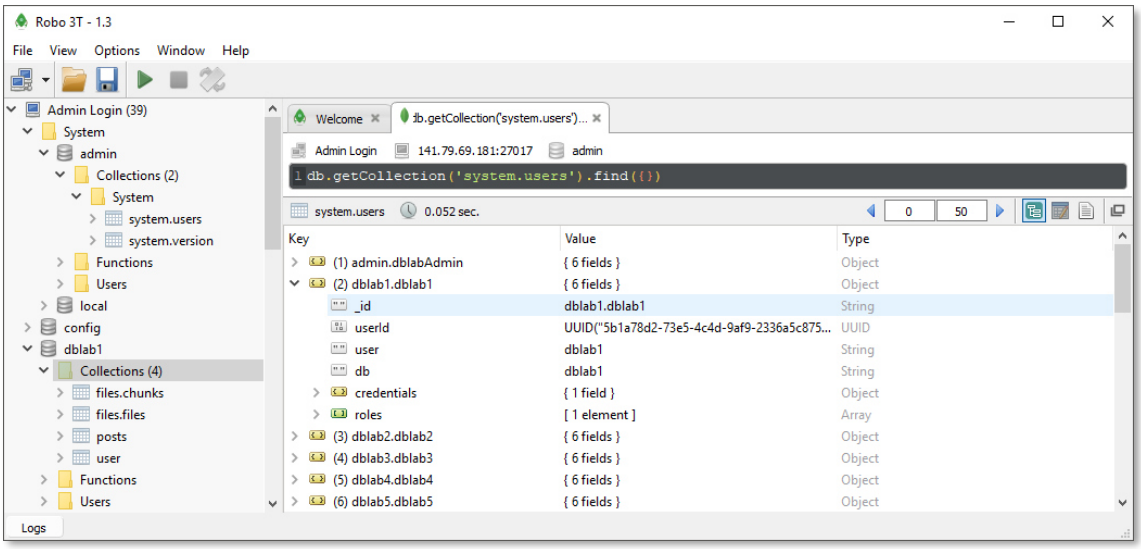


Abbildung 15: Ansicht von Robo 3T

In einer produktiven Umgebung steht die Mongo Shell im Vordergrund, da diese der wichtigste Interaktionspunkt mit der Datenbank ist. Sie ermöglicht eine direkte Interaktion mit der MongoDB auf dem Server. Sie wird auf einem Server durch den Konsolenbefehl *"mongo"* gestartet. Alle weiteren Manipulationen innerhalb der Shell finden mittels einer JavaScript API statt. Es können auch Funktionen direkt in der Shell programmiert und initialisiert werden [21, S. 38-40]. Innerhalb des Labors wird die Shell aber nur zu administrativen Zwecken verwendet.

Wird eine neue MongoDB installiert und gestartet, dann sind einige Datenbanken und Kollektionen bereits enthalten. Die Kollektion denen das Wort *"System"* beigefügt wurde, benutzt die MongoDB zur Verwaltung von Systeminformationen. Unter *"System.users"* werden beispielsweise alle erstellten Benutzer abgespeichert.

Der Start der MongoDB kann mit Parametern versehen werden, beispielsweise die Optionen für die Authentifizierung. Weitere Konfigurationen können das Adressieren eines Replication Sets oder eines Sharded Clusters sein. Dies kann auch durch eine Konfigurationsdatei definiert werden. Wird diese verändert, muss der *"mongod"* Prozess neugestartet werden. Wird eine MongoDB nach einer neuen Installation gestartet, sind keine Benutzer angelegt. Die Authentifizierung sollte somit als Kommandozeilenparameter oder in der Konfigurationsdatei noch nicht festgelegt sein, da sonst keine Authentifizierung möglich ist. Deshalb sollte der erste Schritt nach einer Installation das Anlegen eines Administrativen Benutzers sein [21, S. 142-143]. Danach kann die MongoDB im Authentifikationsmodus neu gestartet werden.

Innerhalb der Shell kann ein Benutzer angelegt werden, indem zuerst die Datenbank ausgewählt wird. Danach kann der Benutzer mit seinem Namen, Passwort und den Rechten eingetragen werden. Die Rechte, die vergeben werden können, lassen sich dabei grob in drei Kategorien gliedern. Die erste sind Rollen, die dem administrativen Bereich zugeordnet werden können. Die zweite sind die Rollen für Benutzer und Anwender einer Datenbank. Die dritte ist für Backups oder Replica Sets. In Bezug auf den Laborversuch wurden alle Datenbanken und Benutzer mit den gleichen Einstellungen angelegt. Die Rolle *"dbOwner"* ist dabei eine Kombination aus Administrator, Benutzerverwaltung und Write/Read-Rechten [22] [21, S. 146-148]. Diese Rolle bezieht sich aber lediglich auf eine einzelne Datenbank innerhalb einer MongoDB.

```
var index;
var maxNumDB = 36
for (index = 1; index <= maxNumDB; index++) {
     db = db.getSiblingDB("dblab"+index);
     db.createUser({
         user:"dblab"+index,
         pwd:"dblab"+index,
         roles:[{
             role:"dbOwner",
             db:"dblab"+index
          }],
         mechanisms:[
          "SCRAM-SHA-1"
         ]})
    db.createCollection("socialNetwork");
    db.createCollection("user");
}
```

Abbildung 16: Eintragen aller Laboruser über die Mongo Shell

5. MongoDB Inhalte im Laborversuch

5.1. Kollektionen und Dokumente

Nachdem die MongoDB gestartet wurde und die Benutzer erstellt worden sind, kann damit begonnen werden, die Daten einzufügen. Die Kollektionen werden in einer MongoDB als erstes angelegt und beinhalten eine Vielzahl an Dokumenten. Alle Kollektionen in einer Datenbank repräsentieren den gesamten Datenbestand. Die Daten über die einzelnen Entitäten werden in einem Dokument abgespeichert. Dieses wird in einer solchen Kollektion gesammelt.

Im Vergleich zu relationalen Datenbanken kann hier der Eindruck einer Parallelität zu Tabellen entstehen. Das ist aber nicht direkt der Fall. Kollektionen enthalten keine Struktur. Eine Kollektion kann die unterschiedlichsten Informationen enthalten. Angenommen eine Kollektion mit der Bezeichnung *"User"* wird erstellt und soll Dokumente mit Benutzerprofilen beinhalten. Ein User Dokument in dieser Kollektion kann verschiedene Informationen besitzen. Diese können sich deutlich von anderen Usern unterscheiden. Ein Benutzer kann seinen Vor- und Nachnamen angeben. Ein anderer möchte nur seinen Vornamen bekannt geben, um nicht die vollständige Identität herzugeben. Ein dritter Benutzer könnte seine Email zur direkten Kontaktaufnahme einfügen [6, S. 30].

Da Dokumente sehr flexibel sind und kein Schema vorgeben, kann in die Kollektion User jedes beliebige Dokument eingefügt werden. Ein Dokument, das die Daten zu einem User wie die eben Beschriebenen enthält, kann umfangreicher sein als das eines anderen. Um dies zu erreichen, verzichtet die MongoDB in ihren Kollektionen auf eine festgelegte Struktur. Ein User Dokument enthält dabei nur die Informationen, die tatsächlich vom User oder dem Anwendungsentwickler erwünscht sind [8, S. 9].

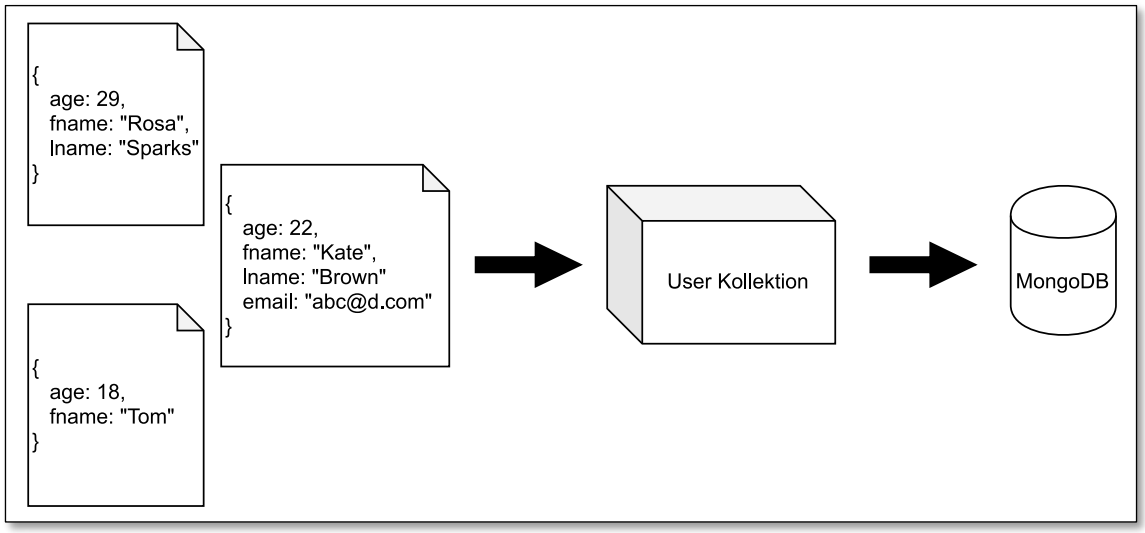


Abbildung 17: Verschiedene Dokumente in einer Kollektion

5.2. Dokumentorientiert mittels JSON

Die Dokumente der Kollektionen sollen im Laborversuch genauer erläutert werden. Hierbei sollten die Möglichkeiten aufgezeigt werden, die dieses flexible Format bietet. Ein wichtiger Bestandteil ist dabei, dass Daten in der Struktur der *"JavaScript object notation"* (JSON) dargestellt werden. Das eigentliche Format, in dem die Daten gespeichert werden, ist aber das *"Binary JSON"* (BSON) Format, einer binären Darstellungsform des JSON Formates. Die Unterschiede, die diese Formate zueinander haben, sollten ebenfalls während des Laborversuchs verdeutlicht werden.

JSON Dateien werden zum Austausch von Daten verwendet. Dies gilt beispielsweise für Schnittstellen, die Informationen zwischen Systemen austauschen. JSON bietet hierbei eine sehr hohe Flexibilität, da im Vergleich zu XML Dateien keine direkte Struktur vorgegeben wird. Es kann eine Vielzahl von Daten strukturiert erfassen und diese gleichzeitig in einem gut leserlichen Zustand abspeichern. Auch kann die Struktur eines JSON Formates deutlich schneller bearbeitet werden [23, S. 221].

Auch relationale Systeme erlangen viele Vorteile, wenn das JSON Format integriert wird. Üblicherweise speichern relationale Systeme Daten ab, die in eine vorgegebene Struktur eingegliedert werden. Unstrukturierte Daten können dabei wegfallen, obwohl diese aus Sicht des Entwicklers von großem Nutzen sein können [24, S. 278]. Hierbei zeigt sich, dass die Vorteile von JSON von vielen Systemen verwendet werden und ein gutes Verständnis für dieses Format somit von großer Bedeutung im Labor ist.

Zu Beginn des Laborversuches, werden deshalb die Kerneigenschaften des JSON Formates vermittelt. Die spezifischen Datentypen des BSON Formates sollen hierbei erläutert und die Unterschiede zu JSON erkenntlich werden. Zusammengefasst sollen die wichtigsten Erkenntnisse innerhalb des Labors sein:

- Verständnis für die JSON Struktur
- Interpretation von Schlüssel/Wert Paaren
- Erkennung und Zuordnung des Datentyps Object (Embedded Document)
- Verwendung von Daten in einem Array
- Die Datentypen im Vergleich zu Binary JSON unterscheiden können

5.3. JSON Architektur

Werden JSON Dokumente angelegt, dann werden die Inhalte in einer einfachen Struktur definiert. Der Anfang des Dokuments beginnt mit zwei geschwungenen Klammern, die das gesamte Dokument repräsentieren. Einzelne Eigenschaften werden als Schlüssel/Wert Paare dargestellt. Ein Schlüssel repräsentiert dabei eine einzelne Eigenschaft und muss innerhalb eines einzelnen JSON Objektes eindeutig sein. Das bedeutet, dass wenn eine Information aus einer JSON Datei ausgelesen werden soll, dann wird der jeweilige Schlüssel angefragt. Das Resultat ist dann der dazugehörige Wert. Schlüsselwerte werden immer in zwei Anführungszeichen angegeben. Dieser wird dann durch einen Doppelpunkt vom Wert getrennt. Der Wert selbst kann dabei in verschiedenen Datentypen eingefügt werden [25, S. 188-190]. Diese sind die folgenden:

- String
- Number
- Object
- Array
- Boolean
- null

Der Datentyp *"Number"* kann dabei drei Arten von Zahlen annehmen. Ganzahlen, Dezimalzahlen und exponentielle Zahlen. Jeweils mit positivem oder negativem Vorzeichen. Dadurch können alle Zahlen als Untermenge dargestellt werden [26, S. 71]. Der Datentyp *"Object"* erlaubt es, eine weitere JSON Struktur unter einem einzelnen Schlüssel unterzubringen. Dadurch können Schlüsselwerte mehrmals vorkommen, ohne dass die Eindeutigkeit beeinträchtigt wird.

```
{
  "fname":"Peter",
  "age" : 25,
  "contact_info":[
      "Hauptstr. 7",
       "peter@aol.de"
  ]
}
```

Abbildung 18: Drei Schlüssel/Wert Paare, einmal mit String, Number und Array

In Abbildung 18 ist ein Beispiel für ein Dokument mit drei Datentypen zu sehen. Hierbei wurden die Datentypen *"String"*, *"Number"* und *"Array"* verwendet. Dies entspricht einem validen JSON Format. Soll nun aus diesem JSON Dokument der Name identifiziert werden, so geschieht dies durch den Schlüsselwert *"name"*. In diesem Fall repräsentiert der Schlüssel einen einzigen Wert. Der Array *"contact_info"* repräsentiert mehrere Werte. Das heißt, dass hier ein Schlüssel mehrere Werte adressiert.

Das Binary JSON Format der MongoDB beinhaltet deutlich mehr Datentypen. Es wurden weitere hinzugefügt, die es ermöglichen, komplexere Daten abzuspeichern. Dazu zählen exemplarisch [27]:

- Binary data
- ObjectId
- Timestamp
- 64-Bit integer
- Date
- Regex

Da JSON nicht alle Datentypen des BSON Formates darstellen kann, wird von der MongoDB eine Erweiterung implementiert, um die Daten zu übertragen. Diese Erweiterung wird auch als *"Extended JSON"* bezeichnet und beinhaltet eine Konvertierung des BSON Formates in ein JSON Format. Hierbei werden grundsätzlich zwei Varianten unterschieden. Die Erste ist der *"Shell mode"*. Dieser zeigt die Daten in der gleichen Art wie die Mongo Shell an. Hierbei werden die BSON Datentypen in einer JSON Struktur dargestellt, wo die neuen Datentypen aber nur die MongoDB Shell und einige Treiber interpretieren können. Dies grenzt den Transport von Daten ein, da viele andere gängige JSON Parser diese Informationen nicht verarbeiten können. Das Format im *"Shell mode"* ist kein valides JSON Format. Deshalb wird hier die zweite Variante, der *"Strict mode"*, verwendet. Dieser wandelt die zusätzlichen Datentypen so um, dass die Datentypen als JSON Format validiert werden können. Um die Daten wieder in das BSON Format umzuwandeln und somit die Datentypen auslesen zu können, müssen die Daten wieder in eine MongoDB integriert werden [28]. In Abbildung 19 sind beide Varianten zu sehen. Es zeigt sich, dass die BSON Datentypen im *"Strict mode"* als neues Schlüssel/Wert Paar dargestellt werden. Der Datentyp selbst wurde dabei als neuer Schlüssel eingefügt, welcher durch ein Dollarzeichen erweitert wurde. Dadurch erkennt die MongoDB einen Datentyp und interpretiert diesen Schlüssel entsprechend.

```
/* Shell mode */
{
  "_id": ObjectId("5ce53df6f0c5f047f5b49e38"),
  "last_login" : ISODate("2019-03-24T23:00:00.000Z")
}

/* Strict mode */
{
  "_id": {
    "$oid": "5ce933d53940b4417fc76b44"
  },
  "last_login": {
    "$date": 1560375232810
  }
}
```

Abbildung 19: Zwei Schlüssel/Wert Paare im Shell und im Strict Mode

5.4. Datenschema validieren

Sollten Daten in unterschiedlichsten Strukturen enthalten sein, kann es sinnvoll sein, vorher einen *"Validator"* zu implementieren. Dieser kann die Struktur eines Dokumentes überprüfen. Innerhalb des Laborversuches wird kein *"Validator"* implementiert. Dennoch wird das Konzept hier erläutert, um zu zeigen, dass trotz der strukturlosen Anordnung von Daten ein festes Schema vorgegeben werden kann.

Als Beispiel wird wieder das User Dokument herangezogen. Beim Erstellen eines neuen User Dokumentes soll das Alter auf die Richtigkeit geprüft werden. Ein Benutzer soll zum Anlegen eines Accounts ein Mindestalter von 16 Jahren angeben. Der *"Validator"* prüft beim Einfügen eines neuen User Dokumentes in eine Kollektion den Wert der angegebenen Felder. Ein JSON Schema wie in Abbildung 20 wird dazu festgelegt, welches alle Informationen zu Validierung beinhaltet [29].

```
db.createCollection("User_Schema", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: [ "age", "fname", "lname"],
            properties:{
                age: {
                    bsonType: "int",
                    minimum: 16,
                },
                fname: {
                    bsonType: "string",
                },
                lname: {
                    bsonType: "string",
                }
            }
        }
    }
})
```

Abbildung 20: Beispiel für einen Validator in der Mongo Shell

```
root@mongo-container2: /
2019-07-17T14:41:14.041+0000 E QUERY    [js] WriteError: Document failed validation :
WriteError({
        "index" : 0,
        "code" : 121,
        "errmsg" : "Document failed validation",
        "op" : {
                "_id" : ObjectId("5d2f338aea4a7afcfc8179f2"),
                "age" : 10,
                "fname" : "Bob",
                "lname" : "Maier"
        }
})
```

Abbildung 21: Eine Fehlermeldung eines nicht validen Dokumentes

5.5. Datenschema der MongoDB

Das flexible Schema zum Anlegen von Daten in der MongoDB, kann ein Vorteil und Nachteil zugleich sein. Die Daten können in unterschiedlicher Struktur angelegt werden und Programmiersprachen können mit den aktuellen Treibern in vollem Umfang auf die Daten zugreifen. Aber damit eine Applikation effizient mit einer MongoDB zusammenarbeiten kann, sollten die Dokumente so gespeichert werden, dass die Daten einer hierfür geeigneten Struktur entsprechen. Das flexible Schema bedeutet nicht, dass kein Schema definiert werden sollte. Der MongoDB Lead Technical Support Engineer William Zola hat in einem Blockbeitrag gesagt:

> *"When designing a MongoDB schema, you need to start with a question that you'd never consider when using SQL: what is the cardinality of the relationship? Put less formally: you need to characterize your "One-to-N" relationship with a bit more nuance: is it "one-to-few", "one-to-many", or "one-to-squillions"? Depending on which one it is, you'd use a different format to model the relationship."* [30]

Das bedeutet, dass vor der Erstellung eines Schemas ermittelt werden sollte, wie die Beziehungen der Daten in der Applikation definiert sind. Das Beispiel mit den User Dokumenten soll hierzu um ein Post Dokument erweitert werden. Eine wichtige Entscheidung könnte sein, wie die Posts und Kommentare der User gespeichert werden. *"Embedded Documents"* können verwendet werden, um Daten innerhalb eines Dokumentes anzulegen. Es könnten aber auch Referenzen gebildet werden, welche auf die Primärschlüssel der User oder der Posts verweisen. Die Aussage von William Zola zeigt, dass die Beziehungen zueinander im Vorfeld definiert werden sollten, um eine sinnvolle Datenstruktur zu erstellen. Um die Beziehungen genauer betrachten zu können, wurde in Abbildung 22 ein vereinfachtes Beispiel eines Social Media Portals erstellt. Dieses zeigt zwei Dokumente. Ein drittes Dokument ist in einem Array eingebettet worden und entspricht einem Kommentar zu einem Post. Hierdurch sollen die beiden Möglichkeiten *"Embedded Documents"* und *"Document References"* sowie die möglichen Kardinalitäten dargestellt werden.

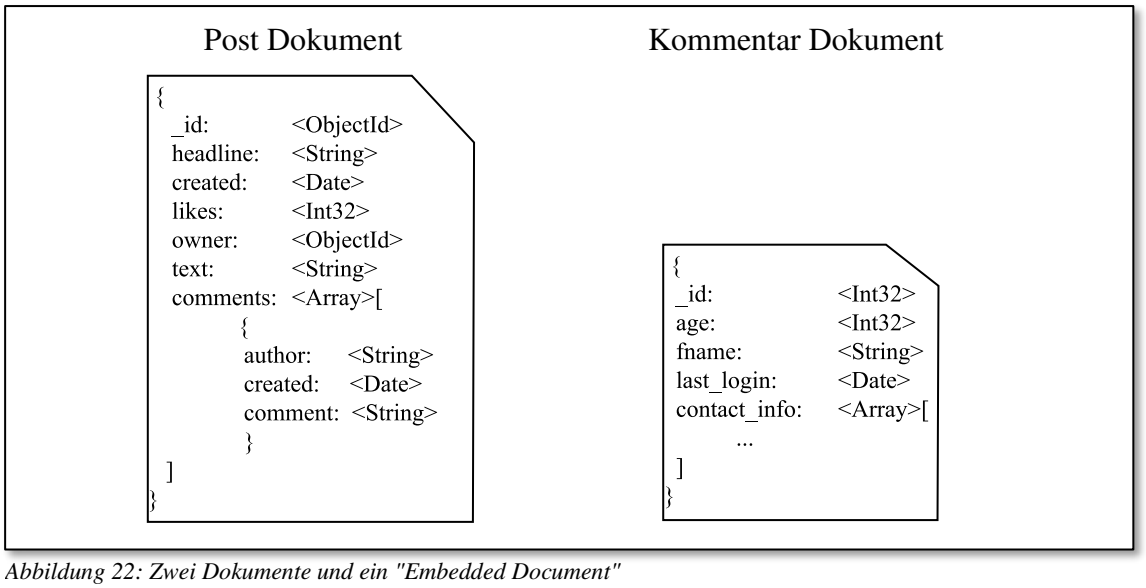


Abbildung 22: Zwei Dokumente und ein "Embedded Document"

5.6. Beziehungen zwischen Daten und Dokumenten

Die Datentypen *"Object"* und *"Array"* haben für die MongoDB eine wichtige Bedeutung. Das Prinzip des *"Embedded Documents"* beruht auf den Datentyp *"Object"*. Dieser ermöglicht es, an einen einzelnen Schlüssel eine Vielzahl von anderen Datentypen unterzubringen. Das heißt ein Schlüssel repräsentiert viele weitere Schlüssel. Ein Array kann mehrere Objekte oder Datentypen beinhalten. Da ein Schlüssel innerhalb eines Dokumentes eindeutig sein muss, kann ein Array hierbei für die nötige Flexibilität sorgen, indem einem Schlüssel viele Dokumente bzw. Objekte zugeordnet werden. Die Adressierung erfolgt dann, durch den Schlüssel des Arrays und des Arrayindexes. Nachfolgend soll erläutert werden, wie die grundlegenden Beziehungen zwischen Dokumenten realisiert werden können.

5.6.1. Embedded Documents

Angenommen ein einzelner Kommentar wird in einem Post Dokument als Objekt angelegt, dann wäre es erforderlich, einen einzigartigen Schlüssel zu generieren. Der Post ist somit innerhalb des Dokumentes eindeutig zu identifizieren, was eine *"Eins-zu-Eins"* Beziehung ermöglicht. Dies wäre in diesem Beispiel aber sehr unpraktisch, da Posts mehrfach vorkommen können und dann alle einen einzigartigen Schlüssel benötigen [31].

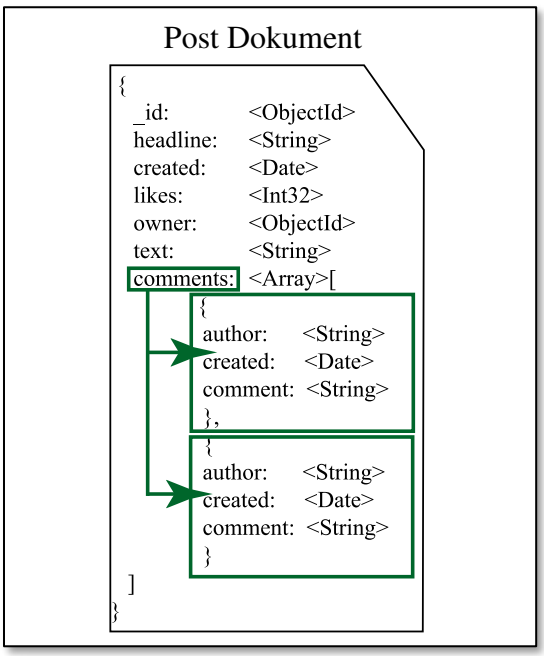


Abbildung 23: Ein Post mit zwei Kommentaren im Array

Für diesen Fall ist die bessere Variante die *"Eins-zu-Viele"* Beziehung. Innerhalb eines Dokumentes kann dies ermöglicht werden, indem ein Array verwendet wird. Der Array selbst wird dann ebenfalls durch einen eindeutigen Schlüssel identifiziert. Dieser kann aber eine Vielzahl an Dokumenten, als Werte enthalten. Das heißt ein Schlüssel kann auf mehrere Dokumente verweisen. Sollen alle Kommentare zu einem Post ausgegeben werden, dann muss lediglich der Array angefordert werden. Dies entspricht der Darstellung in der Abbildung 23. Ein Array im Post Dokument kann somit viele Kommentare enthalten.

Der Vorteil bei *"Embedded Documents"* ist, dass nur eine Suchanfrage gestellt werden muss. Alle Daten sind in einem Dokument enthalten. Das Bearbeiten eines einzelnen Dokuments gilt zudem als atomare Operation. Beachtet werden muss aber, dass mit jedem Post die Dokumentengröße wächst. Die MongoDB speichert Dokumente bis 16MB einzeln ab. Wird dieser Wert überschritten, dann wird das Dokument aufgeteilt. Dies erzeugt Schwierigkeiten bei der Anfrage der Daten [6, S. 38-39].

Die maximale Einbettung von Dokumenten liegt bei einem Limit von 100 Stufen [32]. Dieses Limit ermöglicht die Erstellung von großen Dokumenten. Es macht den Programmieraufwand bei CRUD-Operationen aber umso aufwändiger, da eine sehr komplexe Anfrage an einen großen Datensatz gestellt werden muss. Deshalb kann es von Vorteil sein, mehrere Kollektionen anzulegen, um die Daten leichter zu erfassen.

5.6.2. Document References

Beziehungen über mehrere Kollektionen können über referenzierende Primärschlüssel erstellt werden. Dies stellt die zweite Variante dar, wie Beziehungen zwischen Dokumenten erstellt werden können. Der Primärschlüssel eines Dokumentes wird in den Datenbestand eines anderen integriert, wodurch eine Referenz erzeugt wird [6, S. 40].

Im User Dokument könnte ein Array zu den Posts enthalten sein, welcher die IDs der Posts beinhaltet. Das heißt, das User Dokument referenziert auf mehrere Posts. Hierbei wird ebenfalls eine *"Eins-zu-Viele"* Beziehung erstellt, aber mit dem Unterschied, dass die Dokumente getrennt sind. Es müssten somit zwei Anfragen an die Datenbank gestellt werden. Die Erste, die vom gesuchten User die Post IDs ausliest. Die Zweite, die alle Posts anhand der gefundenen IDs zusammenstellt. Ein negatives Merkmal ist hierbei, dass bei sehr vielen Posts der Array im User Dokument stetig wachsen wird. Das kann wieder zu Problemen mit der maximalen Datengröße führen.

Die bessere Variante ist in Abbildung 24 zu sehen. Hierbei verweisen die einzelnen Posts direkt auf den User. Das heißt der Array im User Dokument fällt weg. Stattdessen wird jedem Kommentar die ID des Posts in einem neuen Feld *"owner"* übergeben. Im Laborversuch wird dieses Prinzip der Referenzierung verwendet, um zu erläutern, wie diese Form der Beziehung hergestellt werden kann.

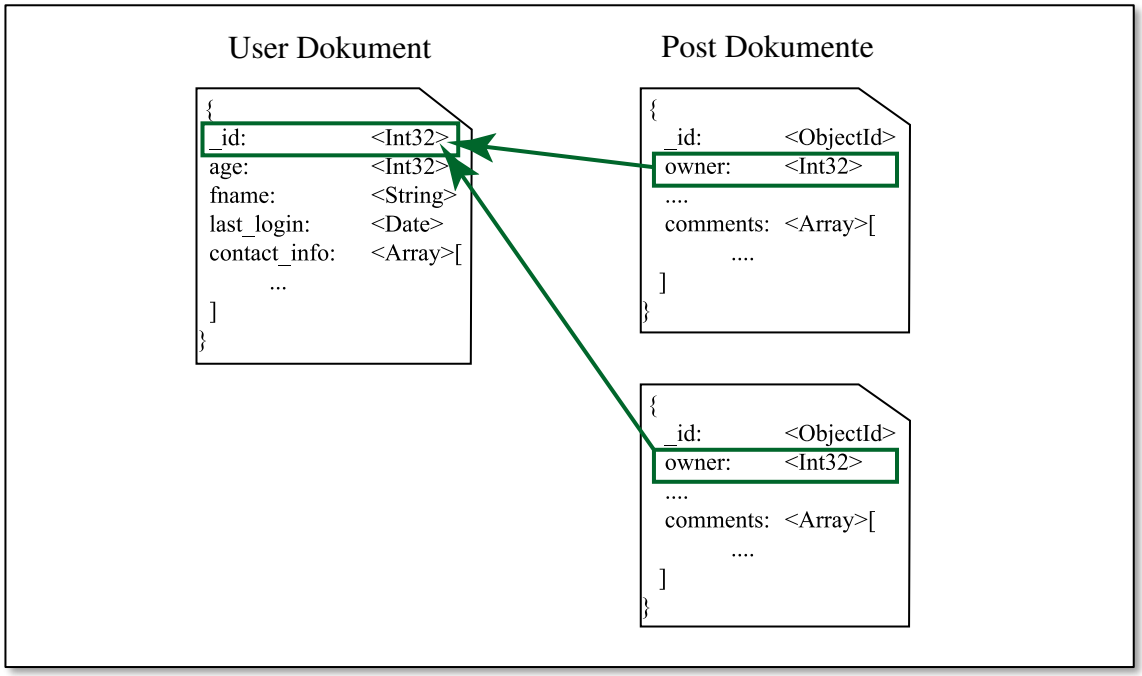


Abbildung 24: Mehrere Post Dokumente referenzieren auf ein User Dokument

5.6.3. Gemischte Beziehungen

Die beiden Möglichkeiten der Beziehungen können auch als hybride Methode verwendet werden. Das heißt ein Dokument kann ein oder mehrere *"Embedded Documents"* enthalten, die wiederum auf andere Dokumente referenzieren. Dadurch werden die Dokumente deutlich komplexer.

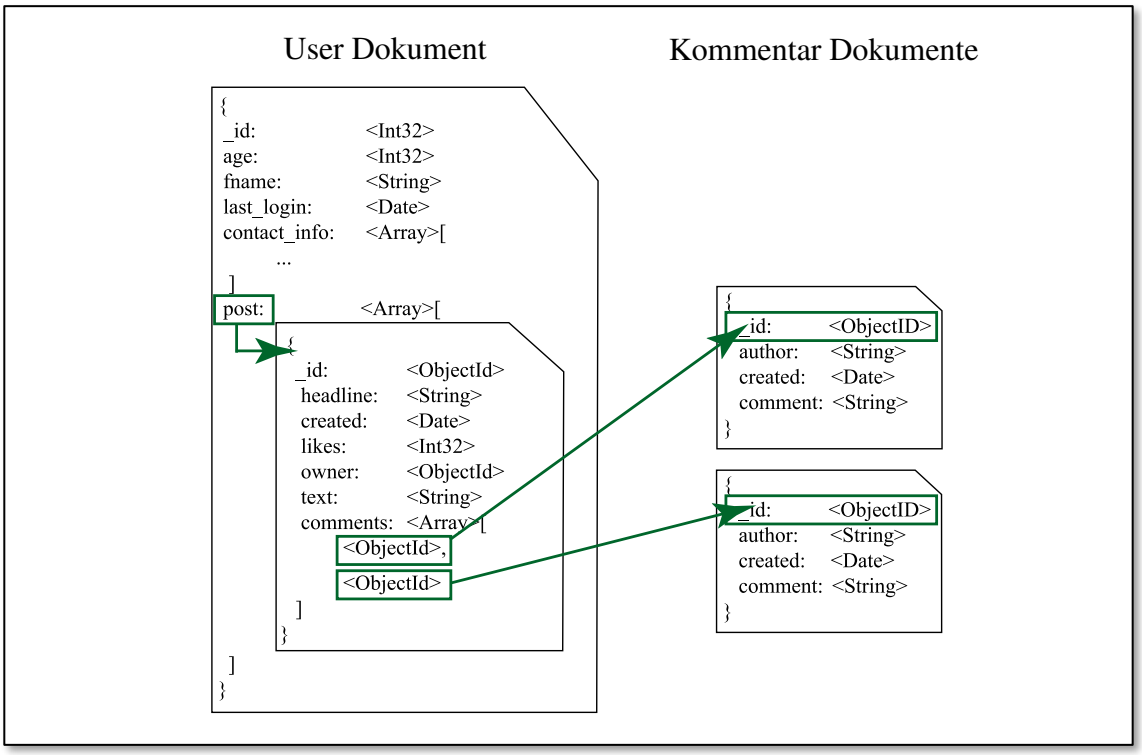


Abbildung 25: Gemischte Beziehungen zwischen Dokumenten

5.6.4. Zwei-Wege Referenzierung

Eine weitere Methode zur Herstellung einer Beziehung kann verwendet werden, wenn zwei Dokumente einander referenzieren sollen. Dies ist ein spezieller Fall, der aber mitunter seine Berechtigung hat. Angenommen das User Dokument würde ergänzt werden und enthält nun einen Schlüssel, der auf das Post Dokument referenziert. Von beiden Dokumenten könnte nun auf das jeweils andere Dokument verwiesen werden. Dies könnte dazu dienen, um eine Anfrage an die Datenbank zu stellen, die zählt, wie viele Posts zu einem bestimmten User gehören. Der Vorteil ist, dass dann nicht mehr beide Kollektionen angefragt werden müssen, um dies zu gewährleisten. Ohne diesen Verweis müsste erst jeder einzelne Post ausgelesen werden und die *"owner"* ID gesucht werden. In diesem Fall wären alle Informationen beim User bereits vorhanden. Der Nachteil ist, dass zum Anzeigen oder ermitteln der Daten eines Posts immer noch das andere Dokument benötigt wird. Ein weiterer Nachteil ist, dass bei einem Update zwei Dokumente betroffen sind, was die Konsistenz gefährden kann [33].

5.7. Mediendaten

Bei der Konzeption einer Datenbank mit der MongoDB sollte noch ein weiterer wichtiger Punkt beachtet werden. Wie bereits erläutert, können Dokumente zwar sehr flexibel gestaltet werden, haben aber eine maximale Dateigröße von 16 MB. Das begrenzt zwar die einzelnen Dokumente in der Größe, hält aber die Performance aufrecht, da kleinere Dokumente sich leichter transportieren lassen [6, S. 24].

Um dennoch Dokumente mit einer größeren Datenmenge speichern zu können, benutzt die MongoDB ein *"Grid File System"* welches als *"GridFS"* bezeichnet wird. Aber nicht nur Dokumente, die zu groß sind, können mit *"GridFS"* in der Datenbank gespeichert werden. Besonders viel Verwendung findet dieses System bei der Speicherung von Mediendaten wie Bilder oder Videos. Die Dokumente werden dabei in kleinere Stücke aufgeteilt. Jedes Einzelne kann dabei eine maximale Größe von 255KB haben. Bei der Verwendung von *"GridFS"* werden zwei Kollektionen von der MongoDB automatisch erstellt [2, S. 178-180].

- *"Files.files"* enthält die Metadaten, welche die Datei beschreiben und die Verbindung zu allen Teilstücken ist.
- *"Files.chunks"* enthält alle Teilstücke aller gespeicherten Daten, Bilder, etc. als Binärdaten.

Die Dokumente die in *"Files.files"* enthalten sind, können zusätzliche Informationen vom Anwender erhalten. Die Datei kann eine Art Beschreibung erhalten. Dadurch lassen sich auf einzelne Abschnitte einer Audiodatei oder eines Videos referenzieren, indem bestimmte *"chunks"* in der Beschreibung festgehalten werden. Dateien müssen nicht in vollem Umfang wiedergegeben werden, nur der referenzierte Bereich kann wiedergegeben werden. Die MongoDB selbst hat kein integriertes Tool, was die Ausführung von *"GridFS"* übernimmt. Die verschiedenen Treiber, wie z.B. Java enthalten Methoden, die es ermöglichen *"GirdFS"* zu verwenden [8, S. 13].

6. Adressierung der MongoDB mit Java

Die MongoDB benötigt im Gegensatz zu DBMS, die mit SQL arbeiten, einen erhöhten Aufwand an Programmierung. Sind die notwendigen Treiber im Java Projekt integriert worden, dann kann eine Verbindung hergestellt werden. In Bezug auf Java soll für Laborteilnehmende ersichtlich sein, welche Methoden und Parameter notwendig sind, um die MongoDB im Authentifikationsmodus zu adressieren und was der Unterschied zwischen den Java Methoden und den MongoDB Operatoren ist. Ziel ist, dass ersichtlich wird, dass die Operatoren der MongoDB verwendet werden, um mit den Daten zu interagieren. Die allgemeine Java Programmierung soll dabei einen geringeren Fokus besitzen. Die MongoDB Java Treiber sollen dazu verwendet werden, um diese Operatoren zugänglich zu machen. Der Programmieraufwand soll deshalb für Teilnehmerinnen und Teilnehmer auf ein Minimum reduziert werden. Dadurch ist gewährleistet, dass der Fokus auf den Inhalten der MongoDB ruht. Es ist nicht das Ziel des Laborversuches, die Java Kenntnisse der Studierenden umfassend zu erweitern.

6.1. Die MongoDB Java Treiber

Die MongoDB bietet Treiber an, die in eine Java Applikation integriert werden können. Grundsätzlich gibt es verschiedene Möglichkeiten, die Treiber einzubringen. Beispielsweise können Build-Management-Tools wie *"Maven"* oder *"Gradle"* verwendet werden. Bei der Verwendung im Labor wird aber auf die herkömmliche Integration der Treiber zurückgegriffen. Das heißt, die JAR Files werden in der Projekt-Struktur hinzugefügt. Dies beinhaltet die JAR Files für die *"BSON Library Driver"*, die *"MongoDB Core Driver"* und die *"MongoDB Sync Driver"* in der Version 3.10.1. Der synchrone Treiber ist abhängig von dem Core und dem BSON Treiber [34]. Das bedeutet, dass für die Vollständigkeit der MongoDB Java Treiber alle drei Java Archive dem Projekt hinzugefügt werden müssen. Wird die Datenbank mit Methoden adressiert, die in den Treibern enthalten sind, kann eine Ähnlichkeit zur Mongo Shell ersichtlich sein. Die Java und JavaScript Syntax der Mongo Shell sind zwar unterschiedlich, aber die grundlegenden Methoden der MongoDB, sind von der Funktion und Benennung fast identisch. In Abbildung 26 ist zu sehen, dass eine Suchanfrage an die Datenbank gleichermaßen aufgebaut ist. In der Mongo Shell wird in der momentanen Datenbank anhand der *"user"* Kollektion eine *"find()"* Methode ausgeführt. In Java läuft dieser Vorgang vergleichbar ab. Eine *"find()"* Methode wird aufgerufen, die anhand eines implementierten Vergleichsdokumentes eine Anfrage ausführt. Festzuhalten ist hierbei, dass die Anfragemethoden vom Aufbau fast identisch sind.

```
/* Mongo Shell JavaScript */
db.user.find({"_id" : ObjectId("5ce0168cfb693a2184d942cb")});

/* Java */
MongoCollection<Document> coll = database.getCollection("user");
Document result = coll.find(
            eq("_id",new ObjectId("5ce0168cfb693a2184d942cb ")
      ).first();
```

Abbildung 26: Eine Suchanfrage in der Mongo Shell und in Java als Vergleich

6.2. Verbindungsaufbau

Der Verbindungsaufbau wird durch das Paket *"com.mongodb.client"* ermöglicht. Eine Instanz dieser Klasse wird angelegt, welche alle Verbindungsinformationen enthält. Über die statische Methode *"create()"* wird dann die Verbindung definiert. Innerhalb der *"create()"* Methode können zwei Parameter definiert werden, die beide eine Verbindung zur MongoDB ermöglichen.

6.2.1. Verbindung mit ConnectionString Klasse

Hierbei werden die Parameter in der Klasse *"ConnectionString"* festgelegt. Wird dieser nicht angegeben, dann wird automatisch eine Verbindung mit Localhost:27017 hergestellt. Im Datenbanklabor wird die Verbindung mit einem Docker Container hergestellt, deshalb muss hier der Pfad und der Port der Docker-Host Maschine als String angegeben werden.

Bei der Herstellung einer Verbindung kann dabei unterschieden werden zwischen dem Verbindungsaufbau mit einer und mehreren Datenbanken. Das bedeutet konkret, dass definiert wird, ob die Applikation mit einem Sharded-Cluster, einem Replication Set oder einer einzelnen Datenbank kommuniziert. Im Falle eines Replication Sets ist nicht notwendig festzulegen, welche Datenbank als Primäre ausgewählt wird [35]. Dies wird durch den Treiber ermittelt, indem ein Ping an alle Datenbanken gesendet wird, und die mit der kürzesten Reaktionszeit ausgewählt wird. Wenn mit einem Replica-Set interagiert wird, kann die Verbindung zu jedem einzelnen Mitglied des Sets überwacht werden. Dazu wird in einem festgelegten Abstand ein *"Heartbeat"* ausgesendet. Dies sind Pings, die in einem regelmäßigen Abstand an die Mitglieder des Sets gesendet werden. Im *"ConnectionString"* können die Parameter für die *"Hearbeats"* eingestellt werden. Der Standardwert liegt bei zwei Sekunden. Erhält eine MongoDB innerhalb von 10 Sekunden keine Rückmeldung auf den Ping, wird die Datenbank als unerreichbar angesehen [36].

6.2.2. Verbindung mit MongoClientSettings Klasse

Diese Klasse ist weit umfangreicher als die Klasse *"ConnectionString"*. Es können eine Vielzahl an Parametern definiert werden, welche gleichzeitig über diverse *"getter"* Methoden wieder ausgelesen werden können. Dadurch wird im Vergleich zum *"ConnectionString"* eine weitaus höhere Interaktionsmöglichkeit mit den Verbindungseinstellungen gewährleistet.

Zur Herstellung einer Verbindung, wird die Klasse *"MongoClientSettings"* verwendet. Die Methode *"builder()"* erhält dabei alle Informationen. Dies sind die Parameter, die für die Herstellung einer Verbindung benötigt werden. Ein Parameter, der enthalten sein muss, ist die Adresse der Docker-Host Maschine mit dem MongoDB Container und der dazugehörige Port. Im Falle eines Replication Sets kann eine Liste von Hosts übergeben werden. Weitere Methoden können die Definition der Berechtigungen enthalten oder die SSL Einstellungen [37].

```
Crede ntial = MongoCredential.createScramSha1Credential(
        "username", "database", password.toCharArray()
);

MongoClient mongoClient = MongoClients.create(
        MongoClientSettings.builder()
           .applyToClusterSettings(
                builder -> builder.hosts(
                        Arrays.asList(
                            new ServerAddress("141.79.69.181", 27017)
                        )
                )
        ).credential(credential).build()
);
```

Abbildung 27: Aufbau einer Verbindung mit den MongoClientSettings

6.3. BSON Dokumente

Das BSON Format der MongoDB kann auch mittels Java Methoden erstellt und bearbeitet werden. Die MongoDB Treiber enthalten dafür eine Bibliothek zu diesem Format. Das Java Archiv *"bson-3.10.1.jar"* enthält alle dafür notwendigen Inhalte. Dabei gibt es aber unterschiedliche Klassen und Interfaces, welche die Möglichkeit anbieten Dokumente mit Hilfe von Java Methoden anzulegen.

Die BSON Library beinhaltet zwei Klassen, um Dokumente in Java anzulegen, die in der MongoDB eingefügt werden können. Die Erste ist die *"org.bson.Document"* Klasse, welche eine Dokument aus Schlüssel/Wert Paaren erstellen kann. Dem Konstruktor kann dabei direkt eine zuvor erstelle Map übergeben werden. Es besteht aber auch die Möglichkeit, mittels der *"append()"* Methode die einzelnen Schlüssel und Werte direkt in das Dokument einzufügen. Die offizielle MongoDB Java Dokumentation bezeichnet diese Klasse als die Bevorzugte, wenn es darum geht Dokumente anzulegen [38].

```
Map<String, Object> userMap = new HashMap<>();
userMap.put("age", 65);
userMap.put("fname", "Dennis");

Document userDoc = new Document(userMap);
```

Abbildung 28: Anlegen eines Dokumentes in Java mit einer Map

```
Document userDoc = new Document();
userDoc.append("age",27);
userDoc.append("fname","Peter");
```

Abbildung 29: Anlegen eines Dokumentes in Java mit der Klasse Document

Das BSON Archiv bietet als zweite Möglichkeit an, die Dokumente als *"BsonDocument"* anzulegen. Diese Klasse erlaubt es, den Datentyp direkt vom Typ BSON einzufügen. Dies ist aber mit einem deutlich höheren Aufwand an Programmierung verbunden. Der Grund hierfür ist, dass diverse *"Codecs"*, *"BsonReader"* und *"BsonWriter"* konstruiert werden müssen, um diese Datentypen verarbeiten zu können [39].

Java Treiber, die weit unter einer Version 3.x lagen, hatten in dem "*MongoDB Core Driver"* eine ältere Möglichkeit enthalten, um Dokumente anzulegen. Das beinhaltete das Interface *"DBObject"* und die Klasse *"BasicDBObject"*. Diese sind in den aktuellen Treibern weiterhin enthalten. Damit wird die Kompatibilität zu älteren Applikationen gewährleistet. Dokumente, die mittels dieses Interfaces angelegt werden, sind oftmals in älteren Literaturen und Beispielprojekten zu finden. Sie gelten aber nicht mehr als die bevorzugte Variante, um Dokumente anzulegen [38].

6.4. Dokumente mit Java anlegen

Das Anlegen eines Dokumentes mit der Klasse *"Document"* soll genauer betrachtet werden. Dazu werden verschiedene Datentypen wie Boolean, Int32, Int64, Object und ein Array eingefügt.

```
Document example = new Document();
example.append("boolean",true);
example.append("int32",122321424);
example.append("int64",46456464654454L);
example.append("SubDoc",new Document("Embedded","Document"));

example.append("Array",Arrays.asList(
       5,
       true,
       "String",
       new Document("Document","in Array")));

example.append("Array Of Documents",Arrays.asList(
       new Document("Document",1),
       new Document("Document",2)));
```

Abbildung 30: Anlegen eines Dokumentes mit verschiedenen Datentypen

In Abbildung 30 wird ein Dokument *"example"* angelegt. Einzelne Werte wie Boolean oder Integer können direkt eingefügt werden. Das heißt, dass in der *"append()"* Methode ein Schlüssel definiert wird, der zugleich den dazugehörigen Wert erhält.

Der Datentypen Objekt wird ebenfalls als Wert eingefügt, allerdings mit dem Unterschied, dass hierbei eine neue Instanz erstellt werden muss. Ein *"Embedded Document"* ist somit eine unabhängige Instanz und kann ebenso komplex aufgebaut sein wie das Dokument, in das es integriert wird.

Ein Array kann alle Datentypen enthalten. Die statische Methode *"asList(array)"* gibt eine Liste basierend auf den gegebenen Parametern zurück [40]. Diese Liste wird in die MongoDB eingefügt und durch die Anordnung im Array mit einem Index versehen. In Abbildung 31 ist ein Dokument zu sehen, das mit dem Java Code aus Abbildung 30 erstellt wurde.

```
{
    "_id" : ObjectId("5cf4e7c1f9865b37ea413161"),
    "boolean" : true,
    "int32" : 122321424,
    "int64" : NumberLong(46456464654454),
    "SubDoc" : {"Embedded" : "Document"},
    "Array" : [
        5, true, "String", {"Document" : "in Array"}
    ],
    "Array Of Documents" : [
        {"Document" : 1},
        {"Document" : 2}
    ]
}
```

Abbildung 31: Ein Dokument, welches aus dem Code in Abbildung 30 erstellt wurde

6.5. Dokumente in Java darstellen

Das Anzeigen eines Dokumentes in Java zeigt zwar die Daten und die JSON Struktur an, entfernt aber die Umbrüche zwischen den Zeilen. Diese sollten aber für das Verständnis der Struktur visualisiert werden. Ein Java Dokument wird mit einer toString() Methode als ein Text interpretiert, der keine Umbrüche oder Einrückungen darstellt. Um dies im Sinne des Laborversuches zu optimieren, wird eine externe Bibliothek integriert, die das Formatieren von JSON Dateien zugänglich macht.

Diese ist die Google GSON Bibliothek, welche es erlaubt, ein JSON Objekt mit allen Lücken und Abständen darzustellen. Dazu wird eine externes JAR File mit der Bezeichnung *"gson-2.8.5.jar*" eingebunden. Google hat bei der Erstellung dieses Archives erzielen wollen, dass Methoden zum Aus- und Einlesen von JSON Dateien mit geringem Aufwand implementiert werden können. Gleichzeitig soll die Struktur der JSON Objekte, unabhängig von der Größe und Komplexität, korrekt visualisiert werden [25, S. 243].

Die Klasse *"Gson"* bietet zwei Methoden an, um JSON Dateien zu serialisieren oder zu deserialisieren. Diese lauten *"toJson()"* und *"fromJson()"*. Diese zeigen oder schreiben ein JSON Objekt mit allen Inhalten an oder speichern Java Objekte als solche ab. Innerhalb des Laborversuches sollen die Dokumente mit allen Umbrüchen und Lücken visualisiert werden. Deshalb wird die *"GsonBuilder"* Klasse hinzugezogen. Diese ermöglicht es einer Instanz der *"Gson"* Klasse Konfigurationen mitzugeben. Die Methode *"setPrettyPrinting()"* kann somit aufgerufen werden und zeigt die Dateien korrekt an [41].

In Abbildung 32 ist zu sehen, dass die *"toString()"* Methode der Klasse *"Document"* alles in einer Zeile ausgibt. Die Ausgabe von GSON ist dabei weitaus anschaulicher.

```
/* Dokument Output */

{"_id": {"$oid": "5ce933d53940b4417fc76b44"}, "age": 23, "fname":
"Katharina", "lname": "Rot", "last_login": {"$date": "2019-05-
11T22:00:00Z"}}

/* GSON Output */

{
  "_id": {
    "$oid": "5ce933d53940b4417fc76b44"
  },
  "age": 23,
  "fname": "Katharina",
  "lname": "Rot",
  "last_login": {
    "$date": "2019-05-11T22:00:00Z"
  }
}
```

Abbildung 32: Unterschied zwischen GSON und toString()

6.6. Mongo CRUD

Die MongoDB bietet eine Vielzahl an Methoden zum Erzeugen, Lesen, Updaten oder Löschen von Daten an. Im Laborversuch sollen die zentralen Methoden erläutert werden, um zu verdeutlichen, wie die MongoDB mit Datensätzen umgeht. Des Weiteren sollen die grundlegenden Operatoren betrachtet werden, um ein Verständnis für die unterschiedlichen Möglichkeiten zum Auslesen und Zusammentragen von Daten zu erhalten. Die allgemeinen Operatoren, die von der MongoDB bereitgestellt werden, sind in den Java Treibern enthalten. Das heißt, ein Verständnis für das allgemeine Vorgehen in Java verdeutlicht somit auch das Verständnis für die MongoDB Interaktion im Allgemeinen.

Unterschieden wird bei den Methoden, ob diese an einem einzelnen Dokument oder an einer Vielzahl von Dokumenten ausgerichtet sind. Die Möglichkeiten lassen sich beispielsweise in eine *"insertOne()"* oder *"insertMany()"* Methode unterteilen.

Filtermethoden

Um die Möglichkeiten der einzelnen CRUD Operationen besser überblicken zu können, wird zuerst die Klasse *"Filters"* der Java Treiber genauer betrachtet. Diese stellt eine Vielzahl von Operatoren bereit, mit denen die Anfragen von Dokumenten gefiltert werden können. Die Filtermethoden in der Klasse entsprechen dabei den MongoDB Operatoren aus der Mongo Shell [21, S. 43-44].

Die einzelnen Filtermethoden werden dabei in folgende Kategorien unterteilt [42]:

- Vergleiche, z.B. *"equals",* welche Werte auf Gleichheit prüft.
- Logisch bedingt, z.B. *"and",* wenn ein Wert mehreren Anfragen entspricht.
- Arrays, z.B. *"size",* wenn ein Array eine bestimmte Länge hat.
- Elements, z.B. *"exists",* wenn ein Dokument ein bestimmtes Feld besitzt.
- Evaluation, z.B. *"regex",* welche nach einer Zeichenkette sucht.
- Bitwise, z.B. *"bitsAllSet",* prüft ob alle Bits in einem Wert auf 0 sind.
- Geospatial, z.B. *"near",* Geodaten die sich in der Nähe einer Position befinden.

```
Document query = coll.find(
       and(
              eq("_id",new ObjectId("5cf913c0e5d1e6d095547485")),
              regex("fname",Pattern.compile("^P.")),
              or(
                     gt("age",30),eq("age",25))
              )
).first();
```

Abbildung 33: Beispiel für eine Anfrage mit Filtermethoden

Create

Die *"insertOne()"* Methode legt ein einzelnes Dokument in der Datenbank an. Wie beim Erstellen der Dokumente bereits zu sehen war, werden die Schlüssel/Wert Paare in einer Instanz der Klasse *"Document"* integriert. Die ObjectId wird dabei von der Datenbank automatisch erzeugt. Wird ein Dokument eingefügt, prüft die MongoDB bei der Ausführung einer "*insert()*" Methode, ob die Ausführung erfolgreich gewesen ist. Dies wird durch die "*Write Concern Specification*" angegeben. Sollte hierbei mit einem Replication Set interagiert werden, dann besteht die Möglichkeit, die Bestätigung von einer sekundären Datenbank zu erhalten. Um dies zu ermöglichen, kann in der Methode definiert werden, von welcher Datenbank eine Bestätigung erfolgen soll. Wird diese Option weggelassen, dann wird automatisch die primäre Datenbank ausgewählt [6, S. 308-309]. Zu beachten ist auch, wenn die angefragte Kollektion, welche von der *"insert()"* Methode adressiert wird, nicht existiert, dann wird eine Neue angelegt. Innerhalb der Mongo Shell wird ein boolescher Wert zurückgegeben und zeigt den Erfolg oder Misserfolg der Methode an. Auch wird die ID des Dokumentes wiedergegeben [21, S. 49-50]. In Java ist dies nicht generell der Fall, hier sind zusätzliche Implementierungen notwendig. Diese wurden innerhalb des Labors nicht vorgenommen, da der Rückgabewert lediglich als Information dient und somit nicht relevant ist.

```
/* Ein Dokument das eingefügt wird */
db.User.insertOne({
   "age":"32",
   "fname":"Peter",
   "lname":"Müller",
   "last_login": new Date()},
{
   w: 1 /*WriteConcern*/
})

/* Rückgabe nach dem Einfügen */
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5cd724a7a231ea485d65a483")
}
```

Abbildung 34: Ein Dokument wird in der Mongo Shell eingefügt

Read

Der Vorgang, der bei der Anfrage eines Datensatzes durchgeführt wird, ist mittels der *"find()"* Methode möglich. Diese gibt alle Dokumente in einer Kollektion wieder. Die *"findOne()"* Methode, welche die gleiche Vorgehensweise bietet, bezieht sich lediglich auf ein einzelnes Dokument. Ein wichtiger Hinweis ist, dass nicht das Dokument selbst zurückgegeben wird, sondern ein *"Cursors"*. Dieser kann als Zeiger angesehen werden, welcher auf den gesuchten Datensatz verweist. Dieser *"Cursor"* kann ein oder mehrere Dokumente zurückgeben. Innerhalb der Mongo Shell muss kein zusätzlicher Schritt unternommen werden, um die ersten 20 Dokumente des Cursors auszugeben [21, S. 41-42]. Sollen mehrere Dokumente ausgegeben werden, dann ist es in Java notwendig, eine Instanz eines Cursors anzulegen. Die *"MongoCursor"* Klasse bietet hier Methoden an, um einzelne oder mehrere Dokumente wiederzugeben. Wird eine Anfrage mit der *"find()"* Methode in Java durchgeführt, dann kann entschieden werden, ob die Methode *"first()"* anhand eines einzelnen oder die Methode *"iterator()"* für mehrere Dokumente angewendet wird.

Um die Suche innerhalb der Kollektion einzugrenzen, kann eine *"Query"* innerhalb der *"find()"* Methode festgelegt werden. Dabei können eine Vielzahl an Filtermethoden verwendet werden, um die Suche auf ein bestimmtes Ziel auszurichten. Einige Beispiele für eine *"find()"* Methode mit integrierter *"Query"* sind in Abbildung 35 zu sehen.

```
/* Mongo Shell JavaScript */
db.user.find( { $and: [
    {"age": { $gt: 50 }},
    {"age": { $lt: 60 }}
]})

/* Java */
Document user = userColl.find(and(gt("age",50),lt("age",60))).first();
```

Abbildung 35: Eine find() Methode in Java und in der Mongo Shell

Wie bereits erläutert wurde, gibt der *"Cursor"* einen Verweis auf das Dokument zurück. Das heißt das gesamte Dokument wird adressiert. Oftmals ist es aber notwendig das Ergebnis einzugrenzen. Nicht alle Schlüssel/Wert Paare sollen angezeigt werden. Innerhalb der *"Query"* kann eine *"Projection"* eingefügt werden, welche nur die definierten Schlüssel/Wert Paare wiedergibt [43].

```
/* Mongo Shell JavaScript */
db.user.find({},{"_id":0,"fname":1,"age":1,"credentials.password":1})

/* Java */
Document user = userColl.find()
       .projection(
              Projections.fields(
                           include("fname","age","credentials.password"),
                           excludeId()
                     )
).first();
```

Abbildung 36: Projektion in Java und der Mongo Shell

Update

Ein Dokument wird ermittelt und dessen Daten verändert oder aktualisiert. Hierbei ist zu beachten, dass unterschiedliche Methoden eingesetzt werden können, um ein Dokument auf einen neuen Stand zu bringen. Dies kann ein Array sein, der um ein weiteres Feld ergänzt wird. Es kann aber auch ein Datum sein, das aktualisiert wird. Die *"updateOne()"* Methode erhält dabei zwei Parameter die durch die MongoDB Operatoren definiert werden [21, S. 52-53]. In Abbildung 37 ist der erste Parameter ein Vergleichsoperator, der das gesuchte Dokument ermittelt. Der zweite Operator setzt ein Datum auf den aktuellen Stand [44].

```
coll.updateOne(eq("fname", "Bobi"),currentDate("last_login"));
```

Abbildung 37: Aktualisierung der Zeit eines Dokumentes anhand der ID

Delete

Die Methoden, mit denen sich Dokumente löschen lassen, sind mit Vorsicht zu verwenden. Diese ermöglichen es einzelne oder eine Vielzahl von Dokumenten aus einer Kollektion zu löschen. Hierbei gibt es keine vordefinierten Sicherheitsmaßnahmen. In Abbildung 38 ist eine *"deleteMany()"* Methode zu sehen, welche eine neue Instanz der Klasse *"Document"* erhält. Dieses Dokument besitzt keinen Inhalt, dennoch führt diese Code Zeile dazu, dass alle Dokumente in einer Kollektion gelöscht werden [45].

```
collection.deleteMany(new Document());
```

Abbildung 38: deleteMany() Methode

Um dies zu verhindern, können den einzelnen Benutzern einer Datenbank selbst definierte Rechte zugeteilt werden. Hierbei kann entschieden werden, ob ein Benutzer das Recht hat, eine Delete Operation durchzuführen. Dadurch kann gewährleistet sein, dass ein Benutzer zwar Lesen, Schreiben und Updates durchführen kann, aber keine Operation zum Löschen von Daten. Dies kann auf jede einzelne Kollektion individuell angepasst werden [46].

Bulk Write

Methoden wie *"updateMany()"*, die es erlauben, mehrere Dokumente gleichzeitig zu bearbeiten, sind wenig flexibel. Ist es erforderlich, dass Dokumente gelöscht und zugleich Neue eingefügt werden, dann kann die Reihenfolge eine wichtige Rolle spielen. Eine *"deleteMany()"* gefolgt von einer *"insertMany()"* Methode kann nicht kombiniert werden. Sie müssen sequenziell ausgeführt werden. Hierbei kann die *"bulkWrite()"* Operation verwendet werden. Diese ermöglicht es, dass Operationen in einer bestimmten Reihenfolge ausgeführt werden, unabhängig davon, welche Operation durchgeführt wird [47].

```
db.User.bulkWrite([
{insertOne :
 {"document":
  {
   "_id":new ObjectId(),"age":NumberInt(19),"fname":"Linda","lname":"Mayer"
  }
 }
},
{updateOne:
 {
  "filter":{"_id":ObjectId("5ce0168cfb693a2184d942d1")},
  "update":{$set:{"age":NumberInt(18)}}
  }
 },
{deleteOne:
 {
  "filter":{"_id":ObjectId("5ce1327ced1e1b3b4f8bf092")}
 }
}
])
```

Abbildung 39: Eine Bulk Write Operation

6.7. Atomare Operationen

Im Jahr 2018 wurde die MongoDB Version 4.0 veröffentlicht. Diese hat die Besonderheit, dass der Funktionsumfang um eine wichtige Komponente erweitert wurde. Es besteht nun die Möglichkeit, atomare Operationen durchzuführen, die mehrere Dokumente in einer oder mehrerer Kollektionen betreffen. Vor dieser Version waren Operationen nur atomar, wenn lediglich ein einzelnes Dokument betroffen war. Beispielsweise ist eine *"bulkWrite()"* Operation nicht atomar, nur die einzelnen darin enthaltenen Operatoren sind es. Werden mehrere Dokumente durch eine solche Operation eingefügt, kann dies bei einem Abbruch dazu führen, dass nur ein Teil der Dokumente eingefügt wurde und der Rest nicht. Komplexe Datenbanken, bei denen eine Vielzahl von Kollektionen verwendet wurden, konnten somit in einen inkonsistenten Zustand geraten. Dies führte dazu, dass bei einigen Anwendungen zusätzlicher Code implementiert werden musste. Dieser sollte Operationen, die mehrere Dokumente betreffen, durch Zwischenergebnisse auf ihre Korrektheit überprüfen. Methoden, die sich auf riesige Datenmengen bezogen, wurden somit in einzelnen Stufen ausgeführt. Das bedeutete aber einen deutlichen Mehraufwand bei der Programmierung [48].

Möglich werden die atomaren Operationen durch das Ausführen von *"Transactions"*. Diese werden innerhalb einer *"Session"* durchgeführt. In Java kann dies ermöglicht werden, wenn aus dem Interface *"ClientSession"* die Methoden *"startTransaction()"* und *"commitTransaction()"* anhand einer *"MongoClient"* Instanz ausgeführt werden. Jeder CRUD Methode wie z.B.*"instertOne()"* wird dabei die *"Session"* als zusätzlicher Parameter übergeben. Eine Operation ist vollständig abgeschlossen, wenn diese durch die *"commitTransaction()"* Methode bestätigt wurde [49]. Die Operationen, die innerhalb einer Transaktion durchgeführt werden, können somit als atomar angesehen werden, wenn die Session bestätigt wird. Ist dies nicht der Fall, dann werden alle Änderungen wieder rückgängig gemacht. Zu beachten ist aber, dass Transaktionen nur in einem *"replica set"* durchgeführt werden können [50].

6.8. Map-Reduce

Eine flexible und effiziente Anwendung, um Daten anzufordern, kann durch die Verwendung des *"Map-Reduce"*-Verfahrens geschehen. Im Laborversuch wird dieses Verfahren wegen seiner Komplexität nicht verwendet. Es sollte dennoch festgehalten werden, was die Eigenschaften dieser Anfragemethode sind und wie es prinzipiell funktioniert. Dieses Verfahren beinhaltet zwei Stufen, die als Ganzes die Zusammenhänge zwischen großen und verteilten Datenbeständen herleiten können. Dabei werden mehrere Kriterien der Funktion zugeführt, nach denen anschließend gefiltert wird. Die Ergebnisse der ersten Stufe werden in einer *"Map"* abgelegt. Nur die Daten, die einem Suchkriterium entsprechen, sind hier eingefügt. Eine Sortierung wird dann anhand dieser Daten durchgeführt. Diese sortierte *"Map"* wird als Zwischenergebnis benutzt, und kann nun einer Reduzierung des Datenbestandes unterzogen werden. In der zweiten Stufe werden alle Werte, die einem Kriterium entsprechen, zusammengefasst [51].

Das *"Map-Reduce"*-Verfahren ist aber mit Bedacht zu verwenden. Sollten komplexe Anfragen an eine Datenbank gestellt werden, ist dieses Verfahren ungeeignet. Das liegt daran, dass die Performance unter Verwendung dieses Verfahrens verringert wird [21, S. 100]. Wird eine *"Map-Reduce"* Funktion an einem Sharded-Cluster angewendet, ist der Aufwand sehr hoch, weil jede Shard die Anfrage bearbeiten muss. Der *"mongos"* Prozess leitet die Anfrage an alle Shards weiter. Jede Einzelne muss die Anfrage komplett abgeschlossen haben. Erst danach sieht der *"monogs"* Prozess die Anfrage als beendet an [52].

In Abbildung 40 ist eine *"Map-Reduce"* Funktion zu sehen. Diese wurde in der Mongo Shell durchgeführt. In der *"Map"* werden alle Posts eines bestimmten Besitzers zusammengetragen. Danach werden in der *"Reduce"* Funktion die Likes zusammengezählt.

```
var mapper = function(){
    if(this.owner == 180001){
        emit(NumberInt(this.owner),NumberInt(this.likes));
    }
};

var reducer = function(key,values){
    var sum=0;
    for(var i in values) sum += values[i];
    return sum;
};

db.posts.mapReduce(
    mapper,
    reducer,
    {out: "user_likes"}
);
```

Abbildung 40: Eine Map-Reduce Funktion, welche die Likes zählt

6.9. Aggregation Pipeline

Anfragen, die komplexer aufgebaut sind, können mit den konventionellen Anfragemethoden wie *"findOne()"* nur schwer hergeleitet werden. Die Verwendung der *"Map-Reduce"* Funktion ist nur unter Beachtung der Performance zu verwenden. Deshalb bietet sich eine weitere Funktion an, die sich besonders gut eignet, um Embedded Documents und Arrays zu bearbeiten. Diese Funktion wird *"Aggregation Pipeline"* genannt. Hierbei können verschiedene Phasen, welche auch als *"Stages"* bezeichnet werden, in einer Anfrage aneinandergehängt werden. Die Aggregation Pipeline ersetzt das Map-Reduce Verfahren nicht, es bietet aber einen anderen Ansatz, um Daten anzufragen. Sie verwendet Operatoren, die von der MongoDB bereitgestellt werden, wohingegen die Funktionen von Map/Reduce in JavaScript selbst geschrieben werden [6, S. 76-77]. Besonders bei Anfragen, die an ein Sharded Cluster gesendet werden, wird eine höhere Performance erzielt. Der Grund dafür ist, dass der *"mongos"* Prozess die Anfrage nur an die Betroffenen Shards weiterleitet, die einen Teil der angefragten Daten enthält [21, S. 88]. Eine einzelne Phase in einem Aggregation Pipeline bildet dabei eine Ausgabe, die an die nächste Phase weitergeleitet wird.

In Abbildung 41 ist ein Beispiel zu sehen. Das Ergebnis der Phase mit dem Operator *"match"* gibt alle Dokumente weiter, welche das Feld *"comments"* enthalten. Hierbei werden noch keine Daten zusammengefasst oder reduziert, denn es findet lediglich ein Vergleich von Werten statt. Die nächste Phase *"unwind"* bezieht sich immer auf ein Feld mit dem Datentyp *"Array"*. Alle Array Felder werden dabei in einzelne Dokumente umgewandelt. Die *"unwind"* Methode benötigt als Parameter einen *"Field Path"*. Das heißt, eine Angabe welches Feld momentan verwendet wird. Dieses wird durch ein Dollarzeichen angegeben. Die Schlüssel/Wert Paare, die nicht innerhalb des Arrays sind, werden ignoriert. Die einzelnen Dokumente, die durch diese Phase erzeugt werden, werden an die Phase mit dem *"Group"* Operator weitergegeben. Diese identifiziert alle Schlüsselwerte mit der Bezeichnung *"$owner"* und zählt diese. Die letzte Phase *"Out"* ist dafür zuständig die Ergebnisse in einer neuen Kollektion abzulegen [21, S. 91]. Zu unterscheiden sind hierbei die Phasen und die Operatoren. Operatoren wie *"Sum",* welcher in Java durch die Klasse *"Accumulators"* aufgerufen werden, sind innerhalb einer *"group"* Phase anzuwenden [53] [21, S. 97].

```
MongoCollection<Document> postCollection = database.getCollection("posts");

AggregateIterable<Document> result  = postsColl.aggregate(
 Arrays.asList(
   Aggregates.match(fields(exists("comments", true))),
   Aggregates.unwind("$comments"),
   Aggregates.group("$owner", Accumulators.sum("Number of comments ", 1)),
   Aggregates.out("allCommentsByUser")
   )
);
```

Abbildung 41: Beispiel für eine Aggregation Pipeline in Java

7. Durchführung des Laborversuches

Das Prinzip des Laborversuches beruht auf der Simulierung einer Social Media Anwendung. Hierzu wurde ein Java Projekt nach dem MVC Prinzip erstellt. Die View enthält dabei alle Elemente, die zur Erstellung des GUIs notwendig sind. Der Controller sorgt für die Einbindung von Methoden zur Interaktion mit dem GUI und dem Model. Das Model selbst sollte die Interaktion mit der Datenbank übernehmen. Hierbei werden aber lediglich vordefinierte Methoden bereitgestellt, welche von den Laborteilnehmenden vervollständigt werden sollen. Durch diese Aufteilung wird bei der Durchführung des Laborversuches der Fokus nur auf die Interaktion mit der Datenbank festgelegt. Zugleich können die Laborteilnehmerinnen und Laborteilnehmer auf den Quellcode zugreifen und sich weiter über das Beispielprojekt informieren. Somit können über die Laboraufgaben hinaus weitere Kenntnisse in Bezug auf die MongoDB erworben werden. Beispielsweise durch eine umfangreiche Modifikation der Laboraufgaben, die mit einer Aggregation Pipeline gelöst werden sollen.

7.1. Die Datensätze in der MongoDB

Die Aufgaben bestehen darin, dass eine bereits integrierte Datenstruktur adressiert wird. Hierzu wird das Social Media Beispielprojekt, das in den vorherigen Kapiteln zur Erläuterung der Beziehung verwendet wurde, erweitert. Diese Datenstruktur besteht aus den Kollektionen *"user"*, *"post"*, *"files.files"* und *"files.chunks"*. Zur Interaktion werden aber lediglich die ersten drei Kollektionen benötigt. Der Grund dafür ist, dass innerhalb einer Aufgabe Mediendaten angefragt werden sollen. Hierzu ist die *"files.chunks"* Kollektion nur für die Speicherung der Daten erforderlich, nicht aber um Anfragen an die Mediendaten zu stellen. In den anderen Kollektionen werden die einzelnen Dokumente dabei so angelegt, dass alle wichtigen Erkenntnisse im Laborversuch vermittelt werden. Die Datenstruktur muss somit die wichtigsten Beziehungen ermöglichen und eine Vielzahl von unterschiedlichen Datentypen beinhalten. In Abbildung 42 ist das Datenschema als JSON Format zu sehen. Die User Dokumente erhalten dabei als einzige nicht die BSON *"ObjectId"* als Primärschlüssel, sondern einen *"Int32"* Wert.

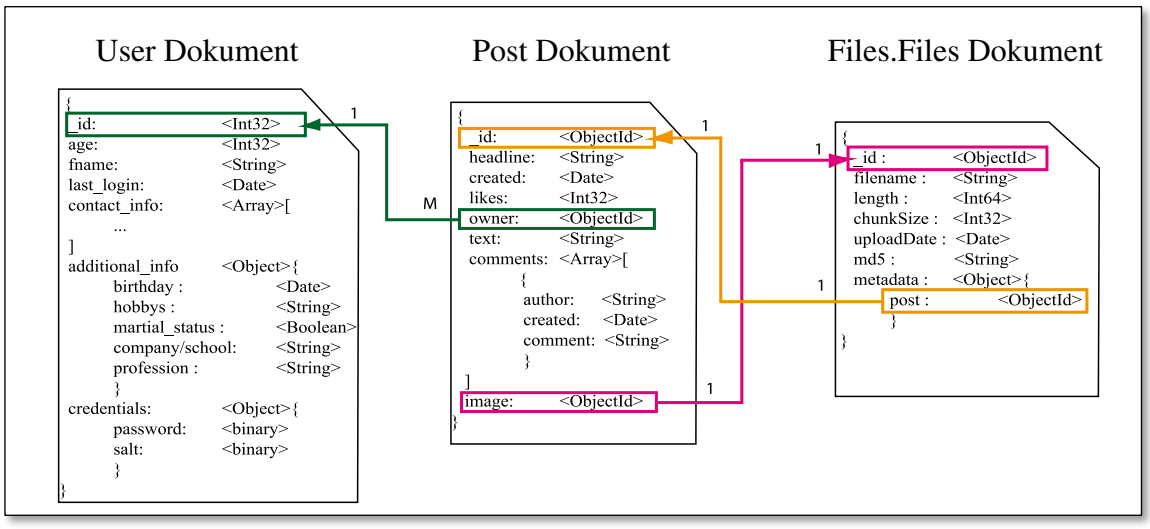


Abbildung 42: Das Datenschema, welches im Laborversuch verwendet wird

User Kollektion
Ein User Dokument beinhaltet den Namen, das Alter, das Datum des letzten Logins, Kontaktinformationen, zusätzliche Informationen über die Person und die Authentifizierungsdaten. Auf den ersten Blick scheint dieses Dokument sehr viele Felder zu beinhalten, aber hierbei muss nochmal darauf hingewiesen werden, dass eine Kollektion keine feste Struktur vorgibt. Diese Vorgabe ist somit nicht zwingend einzuhalten und die Dokumente können unterschiedlich aufgebaut sein. Ein Benutzer muss beispielsweise keine Kontaktinformationen oder zusätzliche Informationen über sich angeben. Auch können von den Laborteilnehmern neue Daten hinzugefügt werden. Dies vorgegebene Sturktur dient lediglich als Orientierung. Beim Vorgehen im Laborversuch können somit die Dokumente erweitert oder mit weniger Daten eingefügt werden. Wichtig ist nur, dass die Lösungen den gegebenen Aufgabenstellungen entsprechen.

Post Kollektion
Ein Post Dokument bietet die selbe Flexibilität wie ein User Dokument und stellt einen klassischen Social Media Post dar. Dieser besitzt beispielsweise die Überschrift, den Text oder das Datum der Veröffentlichung. Ein Post wird von einem User erstellt, was bedeutet, dass es immer einen Bezug zu der Person geben sollte. Ein Post ohne Bezug zum Besitzer würde in einem Social Media Portal als unvollständig angesehen werden. Daten wie der Benutzername, aus einem User Dokument, würden nicht im Post enthalten sein. Es könnten keine Rückschlüsse zum Verfasser des Posts geschlossen werden. Diese Beziehung wird deshalb über den Schlüssel *"owner"* ermöglicht. Viele Posts können von einem Benutzer erstellt werden, aber ein einzelner Post hat nur einen Besitzer.

Auch werden Social Media Posts üblicherweise von anderen Benutzern kommentiert. Dies soll im Beispielprojekt realisiert werden. Dazu werden die Kommentare der anderen Benutzer als *"Embedded Document"* dargestellt. Hierbei wird kein Primärschlüssel benötigt, da durch die Einbettung die Kommentare nur zu einem bestimmten Post gehören können. Ein Kommentar kann sommit nur innerhalb eines Posts existieren.

Files.Files
Ein einzelner Post kann ein Bild enthalten. Da Bilder in einer anderen Kollektion enthalten sind und nicht im Post selbst, referenziert der Schlüssel *"image"* im Post auf die Bild ID. Gleichzeitig enthält ein Bild auch die ID des Posts, zu dem es gehört. Dies stellt eine Zwei-Wege Referenzierung dar. Diese hat den Vorteil, dass die Zugehörigkeit der Bilder unabhängig vom Post festgestellt werden kann.

7.2. Die Java Beispielapplikation

Das Java Projekt, welches die aktuellen Java Treiber für die MongoDB und GSON integriert hat, wird genutzt um mit der MongoDB zu interagieren. Innerhalb des Projektes wurde eine grafische Benutzeroberfläche erstellt. Diese wurde durch die Integration von drei Reitern in Teilbereiche gegliedert. Diese separieren folgende Aufgabenbereiche:

1. JSON Datenstruktur

Das Anlegen und Visualisieren eines einzelnen Dokumentes soll anhand eines Formulars zu Erstellung eines Benutzers geschehen. Hierbei sollen vordefinierte Felder ausgefüllt werden und zugleich die Schlüssel/Wert-Paare mit den unterschiedlichen Datentypen visualisiert werden.

2. Erstellung und Adressierung der Daten mit Java Methoden

Hierbei wurde eine Reihe von Buttons implementiert, von denen jeder einzelne eine Methode ausführt. Zu Beginn des Laborversuches sind die Methoden, die durch diese Buttons adressiert werden, unvollständig implementiert. Dieser Inhalt soll von den Studenten vervollständigt werden. Das Resultat wird in der Benutzeroberfläche angezeigt. Ziel ist es, zu verdeutlichen, wie Datentypen und die Struktur in Java angelegt und angefragt werden können. Auch werden hierbei erste Beziehungen zwischen den Dokumenten durch die Laborteilnehmerinnen und Laborteilnehmer festgelegt.

3. Adressierung von zusammengehörigen Daten

Ein grundlegendes Verständnis für *"Document References"* und *"Embedded Documents"* soll erworben werden. Dazu soll eine Kombination aus allen vorhandenen Kollektionen so adressiert werden, dass ein vollständiger Social Media Post angezeigt wird.

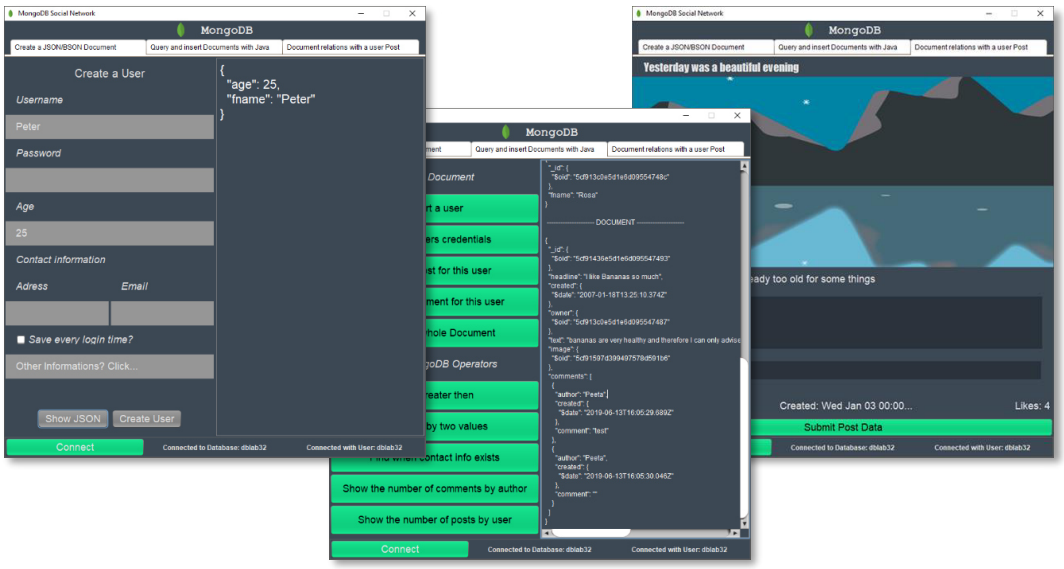


Abbildung 43: Die drei Reiter des Java Beispielprojektes

7.3. Implementierung der Methoden

Alle Implementierungen, die von den Studenten durchgeführt werden, sind in der Klasse *"MongoGUI"* enthalten. Diese Klasse verbindet die Bestandteile des Java Projektes in einer main() Methode. Vom Controller wurde eine anonyme Klasse implementiert, welche die notwendigen Java Methoden zu Interaktion mit der MongoDB überladet. Die einzelnen Methoden repräsentieren dabei zugleich die Aufgaben, die innerhalb des Laborversuches durchgeführt werden sollen.

Die Methoden selbst sind in ihrer Grundform bereits vorhanden. Das bedeutet, dass die Studenten die Methoden nicht neu implementieren werden, sondern den vorhandenen Code mit den wichtigsten Merkmalen ergänzen müssen. Dies soll gewährleisten, dass die Studierenden sich hauptsächlich mit den Anfragen an die Datenbank beschäftigen und weniger mit dem Java Code. Es soll somit eine Trennung zwischen den notwendigen Java Implementierungen zur Ausführung der Methode und den MongoDB Methoden geben.

Als Beispiel können die *"Try"* und *"Catch"* Statements angesehen werden. Diese sind in vielen Methoden, die einen *"MongoCursor"* enthalten, implementiert worden, um das dauerhafte Ausführen einer *"while"* Schleife bei falscher Adressierung zu verhindern. Ein anderes Beispiel ist das Anfordern eines Dokumentes, das ein *"Embedded Document"* beinhaltet. Hierbei soll nicht realisiert werden, dass ein Dokument durch mehrfache Iteration ausgelesen wird. Dies würde einem reinen Java Code entsprechen und sich von den MongoDB Operatoren zu weit entfernen. In Abbildung 44 ist ein Ausschnitt der Methoden zu sehen.

```
public static void main(String [] args){
    MongoView userInterface = new MongoView();
    MongoModel database = new MongoModel();
    MongoController mongoGuiMethods = new MongoController(database,userInterface){


        * Before you start: Configure a connection to MongoDB
       public void createMongoDBConnection(){

        *************************----- Exercise 2: Documents with Java -----******

        * Exercise 2: Create and insert a new document
       public void insertUser(){

        * Exercise 2: Generate password for a user
       public void generateAndInsertPassword(){

        * Exercise 2: A user writes a post
       public void insertPostWrittenByUser(){

        * Exercise 2: Comments on the post
       public void insertComment(){

        * Exercise 2: Show the user and documents from Susanna
       public void showAllInsertedDocuments(){
```

Abbildung 44: Die main-Methode in der Klasse "MongoGUI"

7.4. Beschreibung des Laborversuches

Begleitend zum Laborversuch wurde ein PDF Dokument erstellt, welches alle Grundlagen zur Interaktion mit der MongoDB und dem Beispielprojekt bereitstellt. Dies beinhaltet die Einführung in Robo3T, die Erklärung der Java Klassen und die Aufgabenbeschreibungen. Die Aufgaben als Code sind bereits innerhalb des Java Beispielprojektes enthalten. Das PDF mit den Aufgabenstellungen ist im Anhang enthalten.

In Abbildung 45 ist eine Aufgabenstellung als Beispiel aus dem Java Projekt zu sehen. Daten oder Instanzen, die von den Laborteilnehmerinnen und Laborteilnehmern ergänzt werden müssen, sind entweder durch eine Reihe von Fragezeichen oder einer Null gekennzeichnet.

```
public void insertUser() {

	MongoCollection<Document> userCollection =
		mongoClient.getDatabase().getCollection("?????");

	Document myNewUser = new Document();
	myNewUser.append("_id", 0);
	myNewUser.append("age", 0);
	myNewUser.append("fname","??????????????");
	myNewUser.append("contact_info",
		Arrays.asList("???????????","????????"));
	myNewUser.append("last_login",new Date());

	userCollection.insertOne(myNewUser);
	printJsonResult(myNewUser);
}
```

Abbildung 45: Beispiel für eine Aufgabenstellung für Laborteilnehmende

7.5. Verbindungsaufbau zur MongoDB

Zu Beginn ist es notwendig, die Verbindung zur MongoDB herzustellen. Jede Studentengruppe hat einen Benutzernamen, ein Passwort und eine dazugehörige Datenbank. Ein Beispiel wäre *"dblab1"* als Benutzername und *"dblab1"* als Passwort. Die Datenbank hat ebenfalls den Namen *"dblab1"*.

Innerhalb der Methode "*setMongoDBConnection()*" wurde ein Objekt der Klasse "*ConnectionString*" angelegt. Dieses enthält einen unvollständigen String, welcher mit den Daten der Gruppe ergänzt werden muss. Die grafische Benutzeroberfläche zeigt die erfolgreiche oder fehlgeschlagene Verbindung zur MongoDB an.

7.6. Aufgabenbereich 1: Erstellung von JSON Dokumenten

Ein grundlegendes Verständnis für die Struktur des JSON Formates soll geschaffen werden. Dazu wurde im ersten Reiter ein Formular implementiert, welches die Möglichkeit anbietet, einen neuen Benutzer anzulegen. Die Datenstruktur dieses neuen Benutzers kann vor dem Einfügen in die Datenbank in einem Textfeld betrachtet werden.

Die Methode, welche die Resultate anzeigt, wurde im Beispielprojekt bereits implementiert. Diese wird bei Betätigung des Buttons *"Show JSON"* oder *"Create User"* ausgelöst. Der Button *"Create User"* fügt das Dokument zusätzlich in die Datenbank ein.

Die Aufgabe besteht darin, dass die Studenten die einzelnen Felder ausfüllen und zugleich betrachten, wie das Dokument durch jede einzelne Information erweitert wird. Es soll durch diese Betrachtung ermittelt werden, welche Datentypen integriert werden und in welcher Form dies geschieht. Ein Verständnis für die Datentypen und die Struktur soll geschaffen werden. Das erstellte Dokument kann zuletzt in die MongoDB eingefügt werden. Hierbei soll auch ersichtlich sein, dass der Primärschlüssel automatisch erzeugt wird und sich von den IDs der anderen User unterscheidet. Diese besitzen IDs vom Typ *"String"*, welche explizit bei der Erstellung eines Benutzers angegeben werden muss. Ein Primärschlüssel vom Typ *"ObjectId"* wird von der MongoDB automatisch erzeugt. Eingefügt werden somit folgende Inhalte:

Schlüssel	Datentyp
Vorname	String
Credentials	Object
Password	Binary
Salz	Binary
Alter	Int32
Kontaktinformationen	Array
Adresse, Email	String
Letzter Login	Date
Weitere Informationen	Object
Geburtstag	Date
Hobbys	String
Familienstand	Boolean
Berufsstand	String
Firma/Schule	String

Tabelle 1: Die Datentypen zum User Dokument

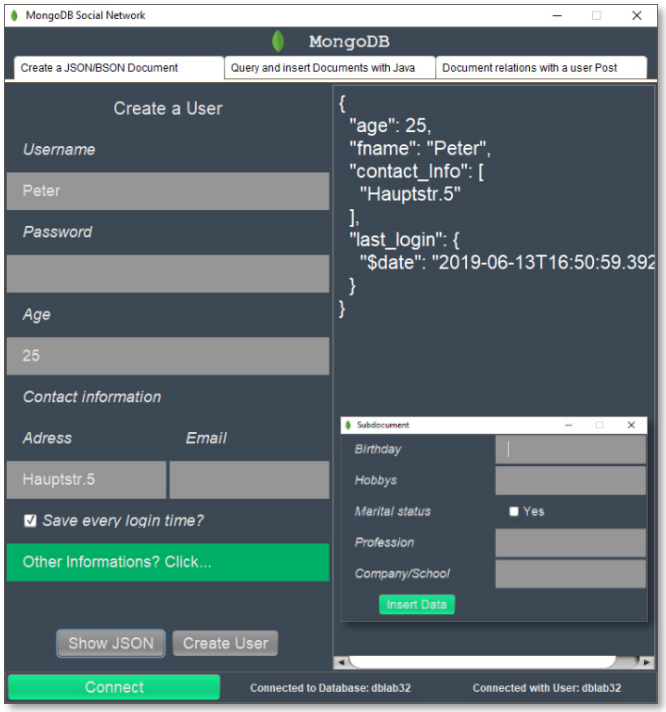


Abbildung 46: Ansicht des ersten Aufgabenbereiches

7.7. Aufgabenbereich 1: Interpretation einer JSON Datenstruktur

Innerhalb dieser Aufgabe sollen die Kenntnisse über das Extended JSON Format erworben werden. Hierbei wird den Laborteilnehmenden eine Struktur eines Dokumentes gezeigt. Die Aufgabe besteht darin, dass die Studenten die Struktur dieses Dokumentes einordnen. Dazu gehört die Erkennung von *"Embedded Documents"* und die BSON Datentypen.

Ziel dieser Aufgabe ist es zu verdeutlicht, wie die MongoDB JSON Dokumente mit den erweiterten Datentypen anlegt. Es sollte den Studierenden bewusst sein, dass die BSON Datentypen in einer validen JSON Struktur dargestellt werden können. Auch sollen die Datentypen *"Array"* und *"Object"* innerhalb des Dokumentes identifiziert werden. Dies soll erläutern, wie Dokumente aufgebaut sind und ein grundlegendes Verständnis für die Datenstruktur schaffen.

Die Abbildung 47 zeigt die Musterlösung des Dokumentes, welches in der Aufgabe schriftlich bearbeitet werden soll. Hierbei ist zu sehen, dass drei Datentypen erkannt und in die richtigen Felder eingefügt werden sollen. Das ist der Datentyp *"ObjectId", "Date"* und *"Binary"*. Beim Datentyp Binary wird eine binäre Zeichenkette mit Base64 codiert und abgespeichert [28]. Zusätzlich wird ein *"Subtype"* gespeichert, der angibt welche Daten abgespeichert werden. Als Standard werden hier allgemeine Binärdaten eingefügt. Das ist durch den *"subType": "00"* zu erkennen [54]. Die Integration dieses Datentyps soll zeigen, dass die BSON Datentypen unterschiedliche Strukturen haben. Des Weiteren sollen ein Array und zwei *"Embedded Documents"* identifiziert werden.

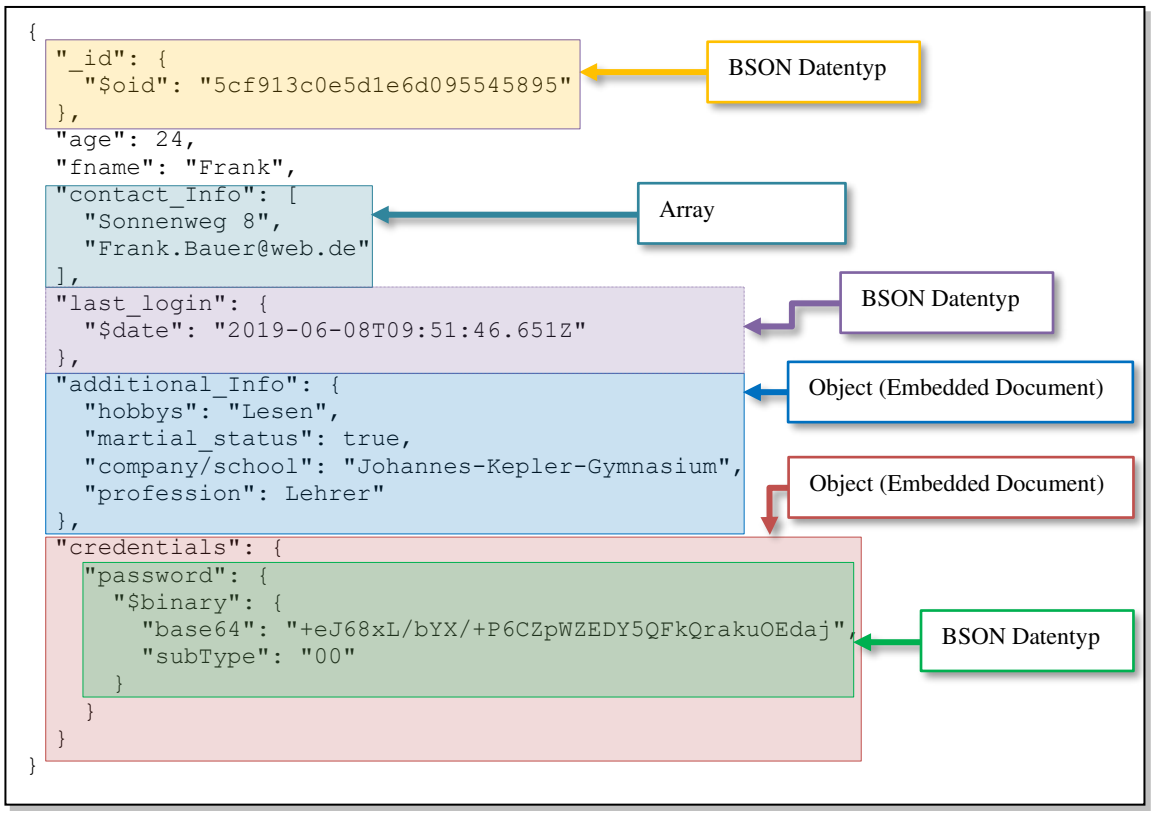


Abbildung 47: Eingliederung der JSON Struktur

7.8. Aufgabenbereich 2: Adressierung der Daten mit Java Methoden

Im zweiten Bereich des MongoDB Laborversuches werden einzelne Methoden im Java Code von den Laborteilnehmenden bearbeitet. Ein Button im GUI führt bei Betätigung eine einzelne Methode aus und zeigt das Resultat in einem Textfeld an. Die verschiedenen Methoden der MongoDB Java Treiber werden verwendet, um Dokumente einzufügen, anzuzeigen und zu manipulieren.

In Abbildung 48 ist der zweite Aufgabenbereich im GUI zu sehen. Der erste Button *"Insert a User"* führt die Methode *"insertUser()"* aus. Das Ergebnis dieser Methode ist auf der rechten Seite als JSON Format zu sehen. Die Datenstruktur wird dabei im *"Strict mode"* angezeigt.

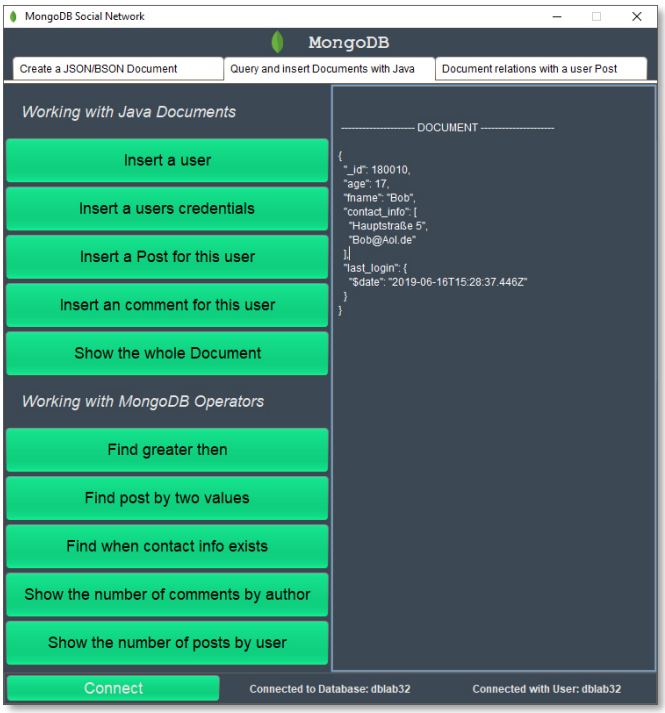


Abbildung 48: Die zweite Aufgabenstellung mit dem Ergebnis der ersten Buttons

Die erste Aufgabe bezieht sich auf die Erstellung und Manipulation eines Dokumentes anhand der Java Methoden. Diese wird im GUI mit der Überschrift *"Working with Java Documents"* dargestellt. Die weiteren Aufgaben sollen die Anfragen von Dokumenten mit Hilfe der MongoDB Operatoren erläutern und einen Ausschnitt der Aggregation Pipeline demonstrieren. Hierbei ist die Überschrift *"Working with MongoDB Operators"* eingefügt worden.

Aufgabe: Neues Dokument erstellen und einfügen

Die Aufgabenstellung ist es, dass ein neues Dokument durch eine Instanz der Klasse *"Document"* angelegt wird. Dieses soll mit Schlüssel/Wert Paaren versehen werden. Die dazu notwendigen Felder werden von den Laborteilnehmenden so ausgefüllt, dass das Resultat der vorgegebenen Aufgabenstellung entspricht. Ziel ist es, folgende Werte einzufügen:

- Die ID (Int32)
- Alter (Int32)
- Vorname (String)
- Kontaktinformationen (Array)
- Die Login Zeit (Date)

Diese Aufgabe soll darstellen, wie ein Dokument in Java erstellt wird und wie spezielle Datentypen eingefügt werden. Es soll auch ersichtlich sein, dass die ID selbst definiert werden kann. Deshalb soll die ID als *"Int32"* angelegt werden und somit identisch zu den anderen Dokumenten aus dem Beispieldatensatz sein.

Um ein Datum einzufügen, könnte die einfachste Herangehensweise sein, dass das aktuelle Datum als String eingefügt wird. Die *"insertOne()"* Methode würde hierbei keine Fehler anzeigen. Dies würde aber nicht der geforderten Aufgabenstellung entsprechen. Es soll explizit ein Datum als Instanz der Klasse "*Date*" eingefügt werden. Hierbei soll ersichtlich sein, dass ein Datentype in die MongoDB eingefügt wird, indem eine Java Klasse verwendet wird. Wird das Dokument nach dem Einfügen in die Datenbank mit den anderen Dokumenten verglichen, so kann ermittelt werden, welche Datentypen eingefügt wurden. Von den Studenten können beliebig viele Dokumente eingefügt werden, bis das zu erzielende Resultat mit korrekten Datentypen vorliegt.

Wird diese Aufgabe mit allen notwendigen Schlüssel/Wert Paaren ausgefüllt, soll bei Betätigung des Buttons *"Insert a User"* die Struktur wie in Abbildung 49 zu sehen sein.

```
{
  "_id": 180010,
  "age": 17,
  "fname": "Bob",
  "contact_info": [
    "Hauptstraße 5",
    "Bob@Aol.de"
  ],
  "last_login": {
    "$date": "2019-06-16T15:14:12.113Z"
  }
}
```

Abbildung 49: Beispielergebnis beim Einfügen eines Dokumentes

Aufgabe: Passwort für einen User generieren

Hierbei besteht die Aufgabenstellung darin, ein *"Embedded Document"* zu erstellen. Dieses soll den Datentyp *"Binary"* beinhalten und zum User Dokument aus der vorherigen Aufgabe hinzugefügt werden. Eine Erkenntnis soll sein, dass innerhalb eines Dokumentes Daten vom Typ *"Binary"* abgespeichert werden können. Zum anderen soll verdeutlicht werden, dass ein Objekt nur integriert werden kann, wenn das richtige User Dokument anhand des Primärschlüssels identifiziert wird.

Das vom Benutzer gewählte Passwort selbst soll als String definiert werden. Ein Array der Klasse *"Byte"* wird angelegt, welcher das Salz enthalten soll, das zur Verschlüsselung des Passwortes benötigt wird. Das Salz wird dazu verwendet, um dem Passwort einen zufällig generierten Wert anzufügen. Die Zeichenfolge, die der Benutzer wählt, wird in Kombination mit dem Salz benötigt, um ein Passwort zu erzeugen, welches vergleichbar zu einer realen Anwendung ist. [55, S. 50-51] Die Methode soll beide Schlüssel/Wert Paare innerhalb einer neuen Instanz der Klasse *"Document"* einfügen. Durch ein Update des User Dokumentes werden die Daten integriert.

Die Aufgabe der Laborteilnehmenden ist es, dass im Java Code das Dokument *"credentials"* mit zwei *"append()"* Methoden so erweitert wird, dass es das Salz und das Passwort beinhaltet. Der richtige Primärschlüssel zur Identifizierung des User Dokumentes kann der vorherigen Aufgabe oder dem Datensatz in Robo3T entnommen werden.

In der Programmierung liegt der Unterschied hierbei beim Einfügen des Dokumentes. Eine *"insertOne()"* Methode wie aus der vorherigen Aufgabe, würde hier ein neues Dokumente erstellen anstatt es einzubetten. Deshalb wird hier die *"updateOne()"* Methode verwendet. Diese enthält einen Operator, der die IDs aller Dokumente vergleicht und durch die Einzigartigkeit des Schlüssels das User Dokument identifiziert. Ist der Schlüssel korrekt, kann das Dokument integriert werden. Das *"Embedded Document"* sollte nach dem Update eine JSON Struktur wie in Abbildung 50 besitzen.


```
{
  "credentials": {
    "password": {
      "$binary": {
        "base64": "poLTGDMGD0WdYShyGzdzwd1tKuPW9FI1fP9ITtBlu5DPU5o
                   Mx+2G9hKWaELd3NpBZVIkatiWcjfyCKbgcZyHMA\u003d\u003d",
        "subType": "00"
      }
    },
    "salt": {
      "$binary": {
        "base64": "yPjtgPoKViWZaF+aSNiM9Q\u003d\u003d",
        "subType": "00"
      }
    }
  }
}
```


Abbildung 50: Ein User Dokument, nachdem das Passwort eingefügt wurde

Aufgabe: Ein User schreibt einen Post

Die Methode *"insertPostWrittenByUser()"* soll ein neues Post Dokument erstellen. Das Post Dokument soll die gleiche Struktur wie die anderen Dokumente in der Kollektion haben. Hierbei soll eine Referenz auf einen User aus dem Beispieldatensatz erzeugt werden. Das User Dokument, das den Namen Susanna beinhaltet, soll mit Hilfe von Robo3T gefunden werden. Diesem Dokument ist der Primärschlüssel zu entnehmen.

Die Aufgabenstellung besteht darin, dass im Java Code ein Dokument mit allen Schlüssel/Wert Paaren versehen wird. Um die Referenz korrekt festzulegen, muss beim Schlüssel *"Owner"* die ID eingetragen werden. Das Ziel ist hierbei, dass verdeutlicht wird, dass ein Wert innerhalb eines Dokumentes auf den Primärschlüssel eines anderen referenziert.

Zu beachten ist hierbei, dass ein neuer Post noch keine Kommentare enthält.


```
{
  "_id": {
    "$oid": "5d09328a5edfb87e98e177f9"
  },
  "headline": "Example",
  "created": {
    "$date": "2019-06-18T18:50:50.416Z"
  },
  "likes": 5,
  "owner": 180004,
  "text": "This is a sample post"
}
```


Abbildung 51: Ein Post Dokument, das eingefügt wurde

Aufgabe: Der Post wird kommentiert

Die Aufgabenstellung ist, den ersten Post, der auf den User mit dem Namen Susanna referenziert, um einen neuen Kommentar zu erweitern. Die Erstellung eines *"Embedded Documents"* ist wieder das Ziel. Der Unterschied liegt aber darin, dass hier der Kommentar in einem Array angelegt wird. Dies soll verdeutlichen, wie vorgegangen werden muss, um eine *"Eins-zu-Viele"* Beziehung innerhalb eines Dokumentes herzustellen.

Hierbei wird die Hilfestellung, die durch den vordefinierten Java Code bislang angegeben war, reduziert. Das Erlernte aus den vorherigen Aufgaben soll nun konkreter genutzt werden. Es werden keine Hinweise zu einem Kommentar gegeben. Die Laborteilnehmenden müssen selbständig den Aufbau eines Kommentars nachvollziehen und mit Hilfe der Java Methoden erstellen.

```
{
  "author": "Rosa",
  "created": {
    "$date": "2019-06-20T16:56:24.836Z"
  },
  "comment": "This is a nice post"
}
```

Abbildung 52: Ein Kommentar, der zu einem bestimmten Post hinzugefügt wurde

Aufgabe: Anzeigen der User und Post Dokumente von Susanna

In dieser Aufgabe soll das Prinzip der Anfrage eines Dokumentes verdeutlicht werden. Es soll das User Dokument mit dem Namen Susanna und alle darauf referenzierenden Posts ausgegeben werden.

Die Aufgabenstellung ist hierbei, dass alle notwendigen Kollektionen definiert werden. Es muss der Primärschlüssel verwendet werden, der in beiden Kollektionen vorkommt. Dies ist die ID des User Dokumentes. Wird die Methode vervollständigt und im GUI durch den Button ausgeführt, dann sollten die IDs in der Ausgabe des Textfeldes überprüft werden. Im Post Dokument sollte als *"owner"* die ID des Users zu finden sein. Sind alle Methoden vollständig und korrekt ausgefüllt, dann sollte nach der Anfrage der Dokumente das Resultat im Textfeld die gleiche Struktur aufweisen wie in Abbildung 53.

Zusätzlich werden Fragen zu den Beziehungen und den Kardinalitäten gestellt. Diese sollten die Laborteilnehmer beantworten.

- Welche Beziehungen und welche Kardinalitäten liegen vor?
  Beziehungen: *"Document References"* oder *"Embedded Document"*?
  Kardinalitäten: 1:1 oder 1:N?

Die Lösungen hierfür sind nachfolgend zu sehen.

- Welche Beziehung besteht zwischen dem User und den Credentials?
  Bezeichnung: Embedded Document
  Kardinalität: 1:1

- Welche Beziehung besteht zwischen dem Post und dem Kommentar?
  Bezeichnung: Embedded Document
  Kardinalität: 1:N

- Welche Beziehung besteht zwischen dem User und dem Post?
  Bezeichnung: Document References
  Kardinalität: 1:N


```
{
  "_id": 180004,
  "age": 17,
  "fname": "Bob",
  "contact_info": [
    "Hauptstraße 5",
    "Bob@Aol.de"
  ],
  "last_login": {
    "$date": "2019-06-15T12:26:09.974Z"
  },
  "credentials": {
    "password": {
      "$binary": {
        "base64": "8GYSdhIF5FWXrHh3hOcejn7zWHlHqnGSg11am26BjDiy5Zyww
                   C6so005MvkPqJymMnBbVnMyTjIVLbfSTKss6A\u003d\u003d",
        "subType": "00"
      }
    },
    "salt": {
      "$binary": {
        "base64": "78QmobsEkkBCQa3DXftrQg\u003d\u003d",
        "subType": "00"
      }
    }
  }
}

{
  "_id": {
    "$oid": "5d04e3e3c49f6c33d7246707"
  },
  "headline": "Example",
  "created": {
    "$date": "2019-06-15T12:26:11.142Z"
  },
  "likes": 5,
  "owner": 180004,
  "text": "This is a sample post",
  "comments": [{
    "author": "Rosa Sparks",
    "created": {
      "$date": "2019-06-15T12:26:12.054Z"
    },
    "comment": "It is a nice post"
  }]
}
```


Abbildung 53: Ein Post Dokument, welches auf ein User Dokument referenziert

Aufgabenbereich 2: Daten mit MongoDB Operatoren erfassen

Hierbei wird ein Ausschnitt der MongoDB Operatoren verwendet, um einige Möglichkeiten aufzuzeigen, wie Daten ausgelesen werden können. Die Operatoren sind wieder in einzelne Methoden separiert worden. Die Aufgabenstellungen beziehen sich auf die Dokumente, die im Rahmen der Beispielapplikation bereits in die Datenbank integriert worden sind. Das Ziel dieser Aufgabe ist es, die grundlegenden Anfragen und Filterungen von Daten zu verdeutlichen.

Aufgabe: Alle User nach Alter filtern

Es sollen die einzelnen Dokumente innerhalb der User Kollektion verglichen werden und nur die Dokumente wiedergegeben werden, die im Feld *"age"* einen größeren *"Int32"* Wert haben als 25. Hierbei wird mit dem *"Greater than"* Selektor interagiert.

Zurückgegeben werden dabei mehrere Dokumente. Zwei wichtige Klassen werden in dieser Aufgabe integriert, um die Interaktion mit der MongoDB genauer darzustellen. Die *"Filters"* Klasse ist im Code der Methode bereits integriert und soll verwendet werden, um zu verdeutlichen wie Operatoren in Java angewendet werden können. Eine Projektion soll nur den Namen des Users ausgeben. Dies soll verhindern, dass das komplette Dokument wiedergegeben wird. Ein Resultat dieser Aufgabe, soll wie in Abbildung 54 aussehen.

```
{
  "_id": 180002,
  "fname": "Olivia"
}

{
  "_id": 180003,
  "fname": "Anton"
}

{
  "_id": 180006,
  "fname": "Olaf"
}

{
  "_id": 180008,
  "fname": "Rosa"
}
```

Abbildung 54: Alle User, die ein größeres Alter als 25 haben

Aufgabe: Verknüpfung von zwei Anfragen

Zwei Operatoren sollen miteinander verbunden werden. Es sollen nur die Dokumente wiedergegeben werden, die beiden Kriterien entsprechen. Die Aufgabe ist es, die Dokumente zu finden, die keine Kommentare beinhalten und die Zeichenkette *"day"* beinhalten. Dies beinhaltet den Logikoperator *"and",* den Elementaroperator *"exists"* und den Evaluationsoperator *"regex"*.

Als Resultat sollte ein einzelnes Dokument angezeigt werden. Dieses ist in Abbildung 55 zu sehen.


```
{
  "_id": {
    "$oid": "5d0ac09e59753bd1fe96b093"
  },
  "headline": "Today is a beautiful day",
  "created": {
    "$date": "2014-01-11T23:00:00Z"
  },
  "owner": 180002,
  "text": "It is beautiful that i can do whatever i want and nobody..."
}
```


Abbildung 55: Ein Dokument, das beiden Anfragen entspricht

Aufgabe: Inhalte überprüfen

Hierbei sollen nur die Dokumente gefunden werden, die mit dem Feld *"contact_info"* versehen sind. Alle die keine Kontaktinformationen besitzen, sollen von der Suchanfrage ausgeschlossen werden. Der Operator *"exists"* wird dazu verwendet, um dies zu ermöglichen. Dieser zeigt alle Dokumente an, die ein bestimmtes Feld besitzen. Eine Boolesche Variable kann zusätzlich festgelegt werden. Ist diese *"true"* dann werden alle Dokumente, die das Feld besitzen, angefordert und zusätzlich alle, die als Wert den Datentyp *"null"* beinhalten. Wird *"false"* eingesetzt, dann werden nur die Dokumente angefordert, die das Feld nicht besitzen [56].

Das Resultat dieser Aufgabe soll mehrere Dokumente anzeigen, wobei alle das Feld *"contact_info"* besitzen sollen. Da hierbei keine Projektion verwendet wird, ist die Ausgabe im Textfeld sehr umfangreich und wird deshalb in Abbildung 56 nur als Ausschnitt gezeigt.

```
"contact_info": [
       "Am Feuerbächl 13"
],

"contact_info": [
       "Herzog-Max-Straße 22"
],

"contact_info": [
       "Hauptstr. 12",
       "Rasa.Parks@aol.com"
],
```

Abbildung 56: Die Kontaktfelder als Resultat der Aufgabe

Aufgabe: Die Anzahl der Posts pro User

Diese Aufgabe soll als Einleitung in die Aggregation Pipeline dienen. Deshalb wird hier nur eine Stage verwendet. Die Aufgabe ist es die *"Group"* Stage so zu vollenden, dass alle Posts nach dem Schlüssel *"Owner"* sortiert werden. Das Resultat in Abbildung 57 zeigt einen Ausschnitt der Ergebnisse an. Jedem Primärschlüssel eines Users wird die Anzahl der Posts zugeordnet.

```
{
  "_id": 180006,
  "Numbers Posts": 2
}

{
  "_id": 180001,
  "Numbers Posts": 2
}

{
  "_id": 180007,
  "Numbers Posts": 2
}
```

Abbildung 57: Die Anzahl der Posts pro User

Aufgabe: Alle Kommentare eines Users finden

Eine weitere Aggregation Pipeline soll verwendet werden, um zu erläutern wie komplexere Anfragen mit mehreren Stages, an die MongoDB gestellt werden können. Hierbei sollen von den Studenten mehrere Stages mit den richtigen Schlüssel/Wert Paaren versehen werden. Die Anfrage, die dabei erstellt wird, gibt die Anzahl der geschriebenen Kommentare innerhalb eines Posts aus.

Hierbei sind drei Stages notwendig, um die Inhalte zu ermitteln. Eine vierte *"out"* Stage gibt das Ergebnis in einer separaten Kollektion aus. Diese dient in diesem Fall lediglich dazu, die Resultate der Pipeline in der Datenbank selbst nochmal zu visualisieren. Das heißt, dass die Ergebnisse als neue Dokumente angelegt werden. Dieser Operator muss immer zum Schluss einer Pipeline angewendet werden und sollte nicht auf bereits existierenden Kollektionen angewendet werden, da diese sonst überschrieben werden [8, S. 199-200].

Stage 1: Aggregates.match()

Gibt nur die Dokumente zurück, die ein Kommentarfeld enthalten. Sollte die Aggregation Pipeline nur diese einzelne Stage enthalten, dann würde das Resultat alle Dokumente ausgeben, die diesem festgelegten Kriterium entsprechen [21, S. 95]. Dies allein ist aber nicht ausreichend, da somit das gesamte Dokument mit allen Daten wiedergegeben wird. Gefordert ist aber nur der Inhalt des Arrays *"comments"*.

Stage 2: Aggregates.unwind()

Um die Suche nur auf die *"Embedded Documents"* einzuschränken, kann die Stage *"unwind"* verwendet werden. Diese erhält immer einen Array als Input, was in dieser Aufgabe das Feld *"comments"* ist. Der Operator erstellt für jeden einzelnen Array Index ein neues Dokument. Hiervon sind aber weiterhin die vollständigen Dokumente mit allen Daten betroffen. Das bedeutet, dass die MongoDB diese Operation nicht ausführen wird, da durch die Aufteilung des Arrays anhand der Kommentare, die ID dupliziert werden würde. Das Dokument würde somit für jeden vorhandenen Kommentar im Array dupliziert werden. Eine ID muss aber eindeutig sein. In Abbildung 58 sind exemplarisch zwei Resultate zu sehen. Die Stage hat für jeden gefundenen Kommentar ein neues Dokument erstellt. Da aber bisher noch keine Filter oder Gruppierungen verwendet wurden, wird der vollständige Inhalt des Input Dokumentes dupliziert. Somit auch die ID.

Abbildung 58: Ein Ausschnitt von zwei Dokumenten mit gleicher ID

Stage 3: Aggregates.group()

Um das Problem mit den redundanten IDs zu lösen, kann eine Gruppierung durchgeführt werden. Diese wird innerhalb des eingebetteten Kommentar Dokumentes anhand des Autors durchgeführt. Die Gruppierung zählt die Anzahl der Werte die zum Schlüssel *"author"* gehören. An die nächste Stage wird dann nur die ID und ein neues Feld mit einem beliebigen Namen weitergegeben, welches die Anzahl beinhaltet [57].

Stage 4: Aggregates.out()

Diese Stage erstellt innerhalb der Datenbank eine neue Kollektion, die das Ergebnis der Aggregation Pipeline beinhaltet. Ein einzelner Datensatz, der von dieser Pipeline ausgegeben wird, sieht wie folgt aus:

```
{
    "_id" : 180001,
    "Number of comments" : 3
}
```

Abbildung 59: Die Anzahl der Kommentare eines Users

Das Ziel dieser Aufgabe ist es, zu verdeutlichen, dass Anfragen mit der Aggregation Pipeline sehr komplex sein können. Anhand des Resultates ist zu sehen, dass Daten innerhalb eines eingebetteten Dokumentes ausgewertet und gruppiert werden können.

7.9. Aufgabenbereich 3: Adressierung eines Social Media Posts

Die bisherigen Aufgaben bezogen sich auf das Erstellen von Dokumenten, das Erstellen von Dokumenten anhand von Beziehungen und das Anfragen von Dokumenten nach vordefinierten Kriterien. In dieser Aufgabe sollen zusammenhängende Daten so adressiert werden, dass ein vollständiges Abbild eines Social Media Posts entsteht. Hierzu müssen alle Kollektionen miteingebunden und die Beziehungen korrekt hergeleitet werden.

Das Ziel dieser Aufgabe ist es, dass die Laborteilnehmenden direkter mit dem Datenbestand interagieren. Anhand des Social Media Posts sollen die Verbindungen zwischen den drei Kollektionen *"Post"*, *"User"* und *"Files.Files"* erläutert werden. Zugleich soll mit dieser Aufgabe ein besserer Bezug zu einem praktischen Beispiel hergeleitet werden.

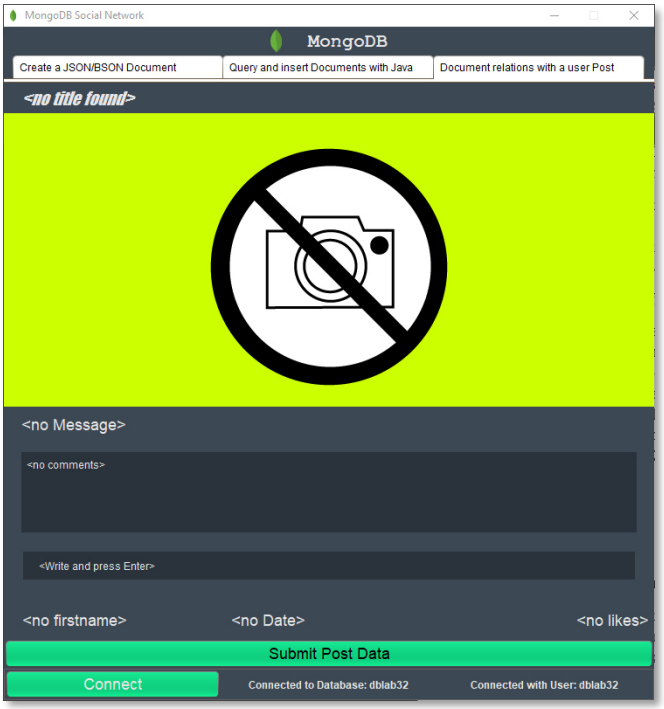


Abbildung 60: Der unvollständige Social Media Post ohne Daten

In Abbildung 60 ist das Grundgerüst des Social Media Posts zu sehen. Die Textfelder sind anfangs noch mit einem Hinweis, wie z.B. *"<no likes>"*, versehen. Ein Bild wurde eingefügt, welches visualisieren soll, dass noch kein Bild aus der Datenbank angefordert wurde.

Ein Post Dokument, das ein Bild enthalten soll, wird von den Studenten ausgewählt. Dazu müssen alle Dokumente in der *"Posts"* Kollektion betrachtet werden, um festzustellen welche Post ein Bild enthält. Die *"Files.Files"* Kollektion, welche die Bilder beinhaltet, muss ebenfalls betrachtet werden, um die ID des Bildes dem Post zuordnen zu können.

Der Dritte Aufgabenteil kann somit in folgende Teilaufgaben untergliedert werden:

- Ein Post Dokument und das dazugehörige Bild in Robo3T identifizieren.
- Die Inhalte des Posts anfordern und auslesen.
- Den Benutzer, der den Post geschrieben hat, identifizieren und den Namen anfordern.
- Das Bild anhand der Referenzierung adressieren und anzeigen lassen.
- Einzelne Schlüssel/Wert Paare in den eingebetteten Kommentaren auslesen.
- Neue Kommentare als *"Embedded Document"* integrieren.

Aufgabe: Anfordern von Daten aus dem Post Dokument

Um diese Aufgabe durchführen zu können, könnte ein Post Dokument wie in Abbildung 61 gewählt werden. Die Datenstruktur in der Abbildung wurde mit Robo3T ermittelt, weshalb diese im *"Shell mode"* angezeigt wird. Dieses Dokument enthält ein Schlüssel/Wert Paar, welches auf ein Image referenziert und eines das auf den Besitzer dieses Posts zeigt. Weiterhin sind zwei Kommentare enthalten.

```
{
    "_id" : ObjectId("5d0ac09e59753bd1fe96b095"),
    "headline" : "Yesterday was a beautiful evening",
    "created" : ISODate("2007-01-03T01:00:00.000+02:00"),
    "likes" : 4,
    "owner" : 180003,
    "image" : ObjectId("5cf916537a8d495ff1aa94a3"),
    "text" : "I think I am probably already too old for some things",
    "comments" : [
        {
            "author" : "Peeta",
            "created" : ISODate("2007-01-06T01:00:00.000+02:00"),
            "comment" : "Yes it was awesome"
        },
        {
            "author" : "Olaf",
            "created" : ISODate("2007-01-06T01:00:00.000+02:00"),
            "comment" : "same here"
        }
    ]
}
```

Abbildung 61: Ein Post Dokument, welches auf ein Bild referenziert.

Zu Beginn werden die Daten angefordert, die direkt aus dem Post Dokument entnommen werden können. Konkret bezieht sich das auf:

- Die Überschrift.
- Die Post Nachricht.
- Die Anzahl der Likes.
- Das Datum.

Die Laborteilnehmenden müssen das Post Dokument anhand des Primärschlüssels mit Hilfe einer *"find()"* Methode anfordern. Die einzelnen Werte können dann über Methoden wie *"getInteger()"* oder *"getString()"* der Klasse *"Document"* angefordert werden.

In der Aufgabe selbst sollen die einzelnen Methoden, welche die Textfelder im GUI adressieren, vervollständigt werden. Dazu müssen die Schlüssel anhand des Post Dokumentes identifiziert und an der richtigen Position eingesetzt werden.

```
"headline" : "Yesterday was a beautiful evening",
"created" : ISODate("2007-01-03T01:00:00.000+02:00"),
"likes" : 4,
"text" : "I think I am probably already too old for some things"
```

Abbildung 62: Die angeforderten Daten aus dem Post Dokument

Aufgabe: Anfordern des User Namens

Der Name der Person, die den Post geschrieben hat, soll angezeigt werden. Dazu werden zwei Kollektionen benötigt. Die Posts Kollektion wird benötigt, um von dem Post den Besitzer zu ermitteln. Die User Kollektion dient dazu, dass die ID des Besitzers mit allen Dokumenten verglichen werden kann. Wird ein Dokument gefunden, dann wird der Schlüssel *"fname"* angefordert.

Zu jedem Post gibt es immer einen einzigen Besitzer. Das heißt, der Name wird korrekt angezeigt, wenn die Laborteilnehmenden die ID des Posts eintragen und den Schlüssel, der auf den User zeigt, adressieren. Der Hintergrund dieser Aufgabe ist, dass die Information des Namens in einem User Dokument gespeichert wird, aber diese Information anhand einer Referenz angefordert werden muss.

Hierbei ist das Ziel zu verdeutlichen, dass nur das Post Dokument anzufragen ist und über die Beziehungen die Inhalte ermittelt werden. Das heißt nur die ID des Posts wird benötigt, alle weiteren Informationen werden durch die Referenzen ermittelt.

```
"fname" : "Anton"
```

Abbildung 63: Der Name des Users, der den Post geschrieben hat

Aufgabe: Integration von Mediendaten

Diese Aufgabe soll einen Einblick in den Umgang mit Mediendaten geben. In diesem Fall sind es Bilder, die im Rahmen des Beispielprojektes bereits in die MongoDB integriert wurden.

Ziel dieser Aufgabe ist zum einen, dass den Laborteilnehmenden vorgeführt wird wie die MongoDB Mediendaten aus dem Datenbestand wiedergeben kann. Zum anderen soll hier wieder mit den Referenzen interagiert werden.

```
"image" : ObjectId("5cf916537a8d495ff1aa94a3"),
```

Abbildung 64: Die ID des Bildes, das zum Post gehört

Aufgabe: Anfordern von eingebetteten Kommentaren

Um alle Kommentare aus dem Post Dokument zu ermitteln, wird eine Aggregation Pipeline verwendet. Zwei Phasen werden innerhalb der Pipeline angelegt. Im Vergleich zur *"Aufgabe 2: Alle Kommentare eines Users finden"* sollen die Kommentare nicht anhand der Autoren gruppiert werden, sondern der Name des Autors und sein geschriebener Kommentar wiedergegeben werden.

```
"comments" : [
        {
            "author" : "Peeta",
            "created" : ISODate("2007-01-06T01:00:00.000+02:00"),
            "comment" : "Yes it was awesome"
        },
        {
            "author" : "Olaf",
            "created" : ISODate("2007-01-06T01:00:00.000+02:00"),
            "comment" : "same here"
        }
    ]
```

Abbildung 65: Alle Kommentare zum Post

Aufgabe: Einen neuen Kommentar einfügen

Um das Prinzip der *"Eins-zu-Viele"* Beziehung besser demonstrieren zu können, soll in dieser Aufgabe eine Methode vervollständigt werden, die neue Kommentare in den Array *"comments"* integriert.

In der Aufgabenstellung selbst werden diesmal keine Hinweise zur Aufgabenlösung gegeben. Das Reduzieren der Hinweise hat den Sinn, dass hier das gesammelte Wissen aus den vorherigen Übungen verwendet werden soll. Der Name, der im Post eingetragen wird, ist abhängig von dem Primärschlüssel den die Laborteilnehmenden auswählen und eintragen.

```
{
       "author" : "Peeta",
       "created" : ISODate("2019-06-28T15:06:21.462+02:00"),
       "comment" : "Das ist ein neuer Kommentar"
}
```

Abbildung 66: Ein neuer Kommentar im Post

Das Resultat der Aufgabe 3

In Abbildung 67 ist ein Post zu sehen. Dies entspricht einem vollständigen Ergebnis der Aufgabe 3. Zu sehen ist, dass die Kommentare alle aus dem Array ermittelt wurden. Werden die Werte mit dem Dokument in Robo3T verglichen, so ist zu sehen, dass der Name des Users mit der ID im Feld *"Owner"* übereinstimmt. Der Zeitpunkt, wo der Post erstellt wurde, wird im Standardformat angezeigt. Hierbei könnte der Wert noch eingegrenzt werden, indem mittels Java Methoden ein passenderes Datumsformat erstellt wird. Da dies aber nicht Bestandteil der MongoDB Methoden ist, wurde dieser Teil ausgelassen.

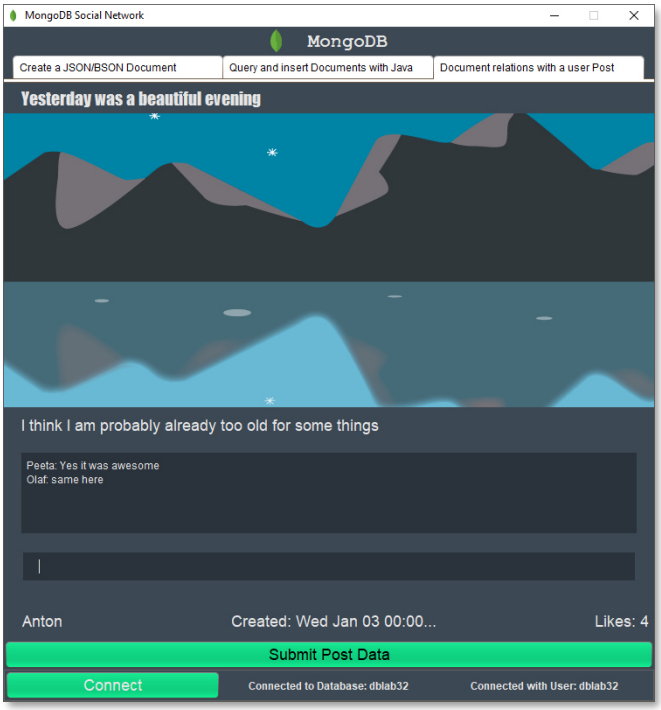


Abbildung 67: Ein vollständig ausgefüllter Post

8. Vorbereitung auf den Laborversuch

Dieser Laborversuch unterscheidet sich sehr von denen, die mit relationalen Systemen zusammenarbeiten. Das liegt zum einen daran, dass alle Klassen, Methoden und Operatoren der MongoDB neu erlernt werden müssen. Die Programmierung von Methoden zur Interaktion mit einer MongoDB ist deutlich zeitintensiver als die Formulierung von SQL Anfragen. Ein weiterer Unterschied bezieht sich auf die Grundlagen der MongoDB. Hierbei wird empfohlen, dass den Studenten vor dem Laborversuch einige theoretische Inhalte vermittelt werden, um den Einstieg in den Laborversuch leichter zu gestalten. Dies bezieht sich auf die folgenden Inhalte:

Primärschlüssel mit dem Datentyp ObjectId
Da der Datentyp *"ObjectId"* durch die hexadezimale Darstellung innerhalb einer einzelnen Kollektion eindeutig ist, sollte dies gegenüber einem Primärschlüssel vom Typ *"Int32"* oder *"String"* verdeutlicht werden. Eine Kollektion kann als Primärschlüssel nur eindeutige Werte vergeben, was bedeutet, dass eine ID vom Datentyp *"String"* auch eindeutig sein muss, aber bei größeren Datensätzen sollte durch die automatische Generierung die ID vom Typ *"ObjectId"* bevorzugt werden. Dies Information lässt sich nur schwer innerhalb des Laborversuches darstellen.

Unterschied BSON und JSON
Das Prinzip des *"Extended JSON"* wird innerhalb des Laborversuches bearbeitet und abgefragt. Dennoch ist es von Vorteil, wenn zuvor einige Informationen bereits bekannt sind. Als wichtige Informationen wird hierbei die Erweiterung der Datentypen durch die Darstellung im JSON Format angesehen. Es sollte vor dem Laborversuch bewusst sein, dass das BSON Format existiert und das es mehr Datentypen beinhaltet als das JSON Format. Dadurch können bei den Laborteilnehmenden Fehlinterpretationen der Ergebnisse verhindert werden.

Embedded Documents
Diese stellen eines der wichtigsten Prinzipen bei der Erstellung von Beziehungen innerhalb der MongoDB dar. Deshalb kann es von Vorteil sein, wenn z.B. eine kurze Einführung in das Prinzip der verschachtelten Daten gibt, damit eingebettete Daten von den Laborteilnehmenden auch ohne viel Aufwand erkannt werden.

Referenced Documents
Dieses Prinzip sollte ebenfalls kurz erläutert werden, damit klar ist, warum mehrere Kollektionen innerhalb des Beispielprojektes verwendet werden. Dadurch sollte die Interaktion mit den einzelnen Dokumenten besser interpretiert werden können.

9. Zusammenfassung und Fazit

Die MongoDB ist momentan eines der meist gefragten DBMS. Die strukturlose Anordnung von Daten und unterschiedliche Einsatzmöglichkeiten sorgen dafür, dass dieses System immer mehr Aufmerksamkeit findet. Zudem kommen immer wieder neue Erweiterungen von den Entwicklern hinzu. Die Wahrscheinlichkeit, dass Studenten der Fakultät M+I nach ihrem Studium häufiger mit einer MongoDB konfrontiert werden, ist somit sehr hoch.

Dadurch das die MongoDB durch die Vielzahl der Treiber in unterschiedlichsten Anwendungen eingesetzt werden kann, können viele Entwickler dieses System in ihre Projekte integrieren. Dabei sind die Grundkenntnisse eine gute Basis, selbst wenn in einem Projekt außerhalb des Datenbank Labors der Hochschule Offenburg mit einer neuen Programmiersprache gearbeitet wird. Durch die Integration des Java Beispielprojektes wird den Studenten zusätzlich die Möglichkeit gegeben, das Projekt in einer eigenen Entwicklungsumgebung auszutesten und zu erweitern. Zugleich muss aber festgehalten werden, dass bei der Interaktion mit der MongoDB ein höherer Schwierigkeitsgrad vorhanden ist als bei den vorherigen Laborversuchen, da die Laborteilnehmenden mit vielen neuen Inhalten konfrontiert werden. Diese erhöhte Schwierigkeit ist aber gerechtfertigt, denn der Einblick in das System ist sehr umfangreich und deckt die wichtigsten Inhalte ab.

Die im Laborversuch integrierten Aufgaben sollen die Grundlagen der MongoDB vermitteln. Hierbei wurde festgestellt das der Programmieraufwand sehr hoch ist, um alle relevanten Informationen zu vermitteln. Die Laboraufgaben in vordefinierten Methoden anzulegen hat dabei den Vorteil, dass sich die Studierenden auf die wesentlichen Inhalte der Datenbank konzentrieren können. Die Java Klassen werden dabei so verwendet, dass die Verständlichkeit des Codes sehr hoch ist. Dies gewährleistet einen guten Überblick der MongoDB Java Klasse, der MongoDB Operatoren und der daraus resultierenden Ergebnisse aus dem Datenbestand.

Auch das Datenschema wurde sehr umfänglich angelegt, damit die gängigsten Datentypen erläutert werden können. Zugleich sollen die wichtigsten Beziehungen zwischen Datensätzen abgebildet werden. Hierbei wurde der Fokus bewusst daraufgelegt, dass die verschiedenen Möglichkeiten des flexiblen Datenschemas verdeutlicht werden. Das Datenschema zeigt somit die Beziehungen zwischen Datensätzen unter Verwendung von *"Embedded Documents"* und *"Referenced Documents"*.

Die Aggregation Pipeline, wurde im Laborversuch nur angerissen, dennoch ist diese eine der wichtigsten und mächtigsten Werkzeuge die dieses DBMS besitzt. Bei einer intensiveren Bearbeitung der MongoDB oder bei einem steigenden Interesse an der Interaktion mit der MongoDB sollte die Aggregation Pipeline einen höheren Fokus bekommen. Die Integration dieser Funktion in den Laborversuch wird als sehr sinnvoll angesehen, da diese sehr viel Verwendung findet und den Umgang mit den Daten der MongoDB deutlich erweitert.

Schlussfolgernd kann gesagt werden, dass durch diese Arbeit und den dazugehörigen Laborversuch alle relevanten Elemente der MongoDB so beleuchtet wurden, dass dieses System gut zugänglich für Studenten ist. Durch das umfangreiche Beispielprojekt können die einzelnen Aufgaben beliebig modifiziert werden. Das heißt für die Zukunft kann anhand dieser Ergebnisse der Laborversuch sehr flexibel angepasst werden. Durch die Verwendung von Docker kann der Laborversuch beliebig modifiziert werden, ohne dass dazu aufwendige Anpassungen am Server notwendig sind. Docker hat somit den Aufbau der MongoDB im Labor deutlich vereinfacht und macht den administrativen Vorgang deutlich effizienter.

10. Literaturverzeichnis

[1] *Worldwide most wanted database knowledge for developers 2018 | Statistic.* [Online] Verfügbar unter: https://www.statista.com/statistics/793854/worldwide-developer-survey-most-wanted-database/. Zugriff am: 22. April 2019.

[2] S. G. Edward und N. Sabharwal, *Practical MongoDB*. Berkeley, CA: Apress, 2015.

[3] J. Batra und S. Batra, „MONGODB Versus SQL: A Case Study on Electricity Data“ in *Emerging Research in Computing, Information, Communication and Applications*, N. R. Shetty, N. H. Prasad und N. Nalini, Hg., Singapore: Springer Singapore, 2016, S. 297–308.

[4] M. Klettke, U. Störl und S. Scherzinger, „Herausforderungen bei der Anwendungsentwicklung mit schema-flexiblen NoSQL-Datenbanken“, *HMD*, Jg. 53, Nr. 4, S. 428–442, 2016.

[5] C. Bach, D. Kundisch, J. Neumann, D. Schlangenotto und M. Whittaker, „Dokumentenorientierte NoSQL-Datenbanken in skalierbaren Webanwendungen“, *HMD*, Jg. 53, Nr. 4, S. 486–498, 2016.

[6] A. Giamas, *Mastering MongoDB 4.x: Expert techniques to run high-volume and fault-tolerant database solutions using mongodb 4.x,* 2. Aufl. Birmingham, UK: Packt Publishing Ltd, 2019.

[7] A. Meier, *Werkzeuge der digitalen Wirtschaft: Big Data, NoSQL & Co: Eine Einführung in relationale und nicht-relationale Datenbanken*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018.

[8] D. Hows, *The definitive guide to MongoDB: A complete guide to dealing with big data using MongoDB,* 3. Aufl. Berkeley CA: Apress, 2015.

[9] D. Swami und B. Sahoo, „Storage Size Estimation for Schemaless Big Data Applications: A JSON-Based Overview“ in *Lecture Notes in Networks and Systems*, Bd. 19, *Intelligent Communication and Computational Technologies: Proceedings of Internet of Things for Technological Development, IoT4TD 2017*, Y.-C. Hu, S. Tiwari, K. K. Mishra und M. C. Trivedi, Hg., Singapore: Springer Singapore, 2018, S. 315–323.

[10] S. Kleuker, *Grundkurs Datenbankentwicklung: Von der Anforderungsanalyse zur komplexen Datenbankanfrage*. Wiesbaden: Springer Fachmedien Wiesbaden, 2016.

[11] S. Bhat, *Practical Docker with Python: Build, Release and Distribute your Python App with Docker*. Berkeley, CA: Apress, 2018.

[12] A. K. Yadav, M. L. Garg und Ritika, „Docker Containers Versus Virtual Machine-Based Virtualization“ in *Advances in Intelligent Systems and Computing, Emerging Technologies in Data Mining and Information Security*, A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya und S. Dutta, Hg., Singapore: Springer Singapore, 2019, S. 141–150.

[13] D. Vohra, *Pro Docker*. Berkeley, California: Apress, 2016.

[14] K. Jangla, *Accelerating Development Velocity Using Docker: Docker Across Microservices*. Berkeley, CA: Apress, 2018.

[15] J. Cook, *Docker for Data Science: Building Scalable and Extensible Data Infrastructure Around the Jupyter Notebook Server*. Berkeley, CA: Apress, 2017.

[16] O. Liebel, *Skalierbare Container-Infrastrukturen: Das Handbuch für Administratoren,* 2. Aufl. Bonn: Rheinwerk Verlag, 2019.

[17] docs.docker.com, *Use bridge networks.* [Online] Verfügbar unter: https://docs.docker.com/network/bridge/. Zugriff am: 18. Mai 2019.

[18] docs.docker.com, *About storage drivers: Sharing promotes smaller images.* [Online] Verfügbar unter: https://docs.docker.com/storage/storagedriver/. Zugriff am: 17. Mai 2019.

[19] docs.docker.com, *Compose file version 3 reference.* [Online] Verfügbar unter: https://docs.docker.com/compose/compose-file/#entrypoint. Zugriff am: 3. Juni 2019.

[20] robomongo.org, *Robo 3T - formerly Robomongo — native MongoDB management tool (Admin UI).* [Online] Verfügbar unter: https://robomongo.org/. Zugriff am: 27. Juni 2019.

[21] D. Bierer, *MongoDB 4 Quick Start Guide: Learn the Skills You Need to Work with the World's Most Popular NoSQL Database*. Birmingham, UK: Packt Publishing Ltd, 2018.

[22] docs.mongodb.com, *Built-In Roles — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/built-in-roles/. Zugriff am: 2. Mai 2019.

[23] A. Grinberg, *XML and JSON Recipes for SQL Server: A Problem-Solution Approach*. Berkeley, CA: Apress, 2018.

[24] D. Petković, „SQL/JSON Standard: Properties and Deficiencies“, *Datenbank Spektrum*, Jg. 17, Nr. 3, S. 277–287, 2017.

[25] J. Friesen, *Java XML and JSON: Document Processing for Java SE,* 2. Aufl. Berkeley, CA: Apress, 2019.

[26] W. Jackson, *JSON Quick Syntax Reference*. Berkeley, CA: Apress, 2016.

[27] docs.mongodb.com, *BSON Types — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/bson-types/. Zugriff am: 22. Mai 2019.

[28] docs.mongodb.com, *MongoDB Extended JSON — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/mongodb-extended-json/. Zugriff am: 22. Mai 2019.

[29] docs.mongodb.com, *Schema Validation — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/core/schema-validation/. Zugriff am: 18. Mai 2019.

[30] William Zola, *6 Rules of Thumb for MongoDB Schema Design: Part 1 | MongoDB Blog.* [Online] Verfügbar unter: https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1. Zugriff am: 24. Mai 2019.

[31] docs.mongodb.com, *Model One-to-One Relationships with Embedded Documents — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/. Zugriff am: 24. Mai 2019.

[32] docs.mongodb.com, *MongoDB Limits and Thresholds — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/limits/. Zugriff am: 24. Mai 2019.

[33] William Zola, *6 Rules of Thumb for MongoDB Schema Design: Part 2 | MongoDB Blog.* [Online] Verfügbar unter: https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2. Zugriff am: 25. Mai 2019.

[34] mongodb.github.io/mongo-java-driver/, *Installation.* [Online] Verfügbar unter: http://mongodb.github.io/mongo-java-driver/3.10/driver/getting-started/installation/. Zugriff am: 12. Mai 2019.

[35] mongodb.github.io/mongo-java-driver/, *Connect to MongoDB.* [Online] Verfügbar unter: http://mongodb.github.io/mongo-java-driver/3.10/driver/tutorials/connect-to-mongodb/. Zugriff am: 18. April 2019.

[36] mongodb.github.io/mongo-java-driver/, *ConnectionString.* [Online] Verfügbar unter: http://mongodb.github.io/mongo-java-driver/3.10/javadoc/com/mongodb/ConnectionString.html. Zugriff am: 4. Mai 2019.

[37] mongodb.github.io/mongo-java-driver/, *MongoClientSettings.* [Online] Verfügbar unter: http://mongodb.github.io/mongo-java-driver/3.10/javadoc/com/mongodb/MongoClientSettings.html. Zugriff am: 5. Mai 2019.

[38] mongodb.github.io/mongo-java-driver/, *Documents.* [Online] Verfügbar unter: https://mongodb.github.io/mongo-java-driver/3.10/bson/documents/. Zugriff am: 2. Juni 2019.

[39] mongodb.github.io/mongo-java-driver/, *Readers and Writers.* [Online] Verfügbar unter: https://mongodb.github.io/mongo-java-driver/3.10/bson/readers-and-writers/. Zugriff am: 27. Juni 2019.

[40] docs.oracle.com, *Arrays (Java Platform SE 7).* [Online] Verfügbar unter: https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#asList(T…). Zugriff am: 3. Juni 2019.

[41] javadoc.io, *Gson 2.8.5 API.* [Online] Verfügbar unter: https://www.javadoc.io/doc/com.google.code.gson/gson/2.8.5. Zugriff am: 25. Mai 2019.

[42] mongodb.github.io/mongo-java-driver/, *Filters.* [Online] Verfügbar unter: https://mongodb.github.io/mongo-java-driver/3.10/builders/filters/. Zugriff am: 8. Juni 2019.

[43] docs.mongodb.com, *Project Fields to Return from Query — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/tutorial/project-fields-from-query-results/#projection-document. Zugriff am: 8. Juni 2019.

[44] docs.mongodb.com, *$currentDate — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/operator/update/currentDate/. Zugriff am: 17. Juli 2019.

[45] docs.mongodb.com, *Delete Documents — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/tutorial/remove-documents/index.html. Zugriff am: 8. Juni 2019.

[46] docs.mongodb.com, *Collection-Level Access Control — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/core/collection-level-access-control/. Zugriff am: 8. Juni 2019.

[47] docs.mongodb.com, *db.collection.bulkWrite — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/. Zugriff am: 28. Juni 2019.

[48] Mat Keep und Alyson Cabral, *MongoDB Multi-Document ACID Transactions are GA | MongoDB Blog.* [Online] Verfügbar unter: https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability. Zugriff am: 18. Mai 2019.

[49] mongodb.github.io/mongo-java-driver/, *ClientSession.* [Online] Verfügbar unter: https://mongodb.github.io/mongo-java-driver/3.10/javadoc/com/mongodb/client/ClientSession.html. Zugriff am: 9. Juni 2019.

[50] docs.mongodb.com, *Transactions — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/core/transactions/. Zugriff am: 1. Mai 2019.

[51] docs.mongodb.com, *Map-Reduce — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/core/map-reduce/. Zugriff am: 26. April 2019.

[52] docs.mongodb.com, *Map-Reduce and Sharded Collections — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/core/map-reduce-sharded-collections/. Zugriff am: 2. Mai 2019.

[53] mongodb.github.io/mongo-java-driver/, *Accumulators.* [Online] Verfügbar unter: https://mongodb.github.io/mongo-java-driver/3.10/javadoc/com/mongodb/client/model/Accumulators.html. Zugriff am: 14. Juni 2019.

[54] bsonspec.org, *BSON (Binary JSON): Specification Version 1.1.* [Online] Verfügbar unter: http://bsonspec.org/spec.html. Zugriff am: 20. Juni 2019.

[55] S. Haunts, *Applied Cryptography in .NET and Azure Key Vault*. Berkeley, CA: Apress, 2019.

[56] docs.mongodb.com, *$exists — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/operator/query/exists/#op._S_exists. Zugriff am: 16. Juni 2019.

[57] docs.mongodb.com, *$group (aggregation) — MongoDB Manual.* [Online] Verfügbar unter: https://docs.mongodb.com/manual/reference/operator/aggregation/group/. Zugriff am: 16. Juni 2019.

11. Abbildungsverzeichnis

Abbildung 1: Schema des BSON und des JSON Formates [9, S. 319] 7
Abbildung 2: MongoDB Kernprozesse anhand von zwei vernetzten Datenbanken 8
Abbildung 3: Visualisierung der Datenspeicherung in einem MongoDB Replica-Set ...10
Abbildung 4: Beispiel-Code für das Auslesen des Datums aus einer ObjectID in Java .13
Abbildung 5: Beispiel-Ausgabe in Java anhand des Datums einer ObjectID 13
Abbildung 6: Virtuelle Maschine vs. Docker [12, S. 147] 16
Abbildung 7: Download des offiziellen MongoDB Images 17
Abbildung 8: Docker-Container wird mit Port gestartet 18
Abbildung 9: Docker History des offiziellen MongoDB Image 19
Abbildung 10: Die History des importierten MongoDB Containers 19
Abbildung 11: Beispiel für ein Dockerfile 20
Abbildung 12: Build Befehl zum Erstellen eines Images aus einem Dockerfile 20
Abbildung 13: Erstellen eines Images und eines Containers 20
Abbildung 14: Docker-Compose File für die MongoDB im Labor 23
Abbildung 15: Ansicht von Robo 3T 24
Abbildung 16: Eintragen aller Laboruser über die Mongo Shell 25
Abbildung 17: Verschiedene Dokumente in einer Kollektion 27
Abbildung 18: Drei Schlüssel/Wert Paare, einmal mit String, Number und Array 29
Abbildung 19: Zwei Schlüssel/Wert Paare im Shell und im Strict Mode 30
Abbildung 20: Beispiel für einen Validator in der Mongo Shell 31
Abbildung 21: Eine Fehlermeldung eines nicht validen Dokumentes 31
Abbildung 22: Zwei Dokumente und ein "Embedded Document" 32
Abbildung 23: Ein Post mit zwei Kommentaren im Array 33
Abbildung 24: Mehrere Post Dokumente referenzieren auf ein User Dokument 34
Abbildung 25: Gemischte Beziehungen zwischen Dokumenten 35
Abbildung 26: Eine Suchanfrage in der Mongo Shell und in Java als Vergleich 37
Abbildung 27: Aufbau einer Verbindung mit den MongoClientSettings 39
Abbildung 28: Anlegen eines Dokumentes in Java mit einer Map 40
Abbildung 29: Anlegen eines Dokumentes in Java mit der Klasse Document 40
Abbildung 30: Anlegen eines Dokumentes mit verschiedenen Datentypen 41
Abbildung 31: Ein Dokument, welches aus dem Code in Abbildung 30 erstellt wurde .41
Abbildung 32: Unterschied zwischen GSON und toString() 42
Abbildung 33: Beispiel für eine Anfrage mit Filtermethoden 43
Abbildung 34: Ein Dokument wird in der Mongo Shell eingefügt 44
Abbildung 35: Eine find() Methode in Java und in der Mongo Shell 45
Abbildung 36: Projektion in Java und der Mongo Shell 45
Abbildung 37: Aktualisierung der Zeit eines Dokumentes anhand der ID 46
Abbildung 38: deleteMany() Methode 46
Abbildung 39: Eine Bulk Write Operation 47
Abbildung 40: Eine Map-Reduce Funktion, welche die Likes zählt 49
Abbildung 41: Beispiel für eine Aggregation Pipeline in Java 50
Abbildung 42: Das Datenschema, welches im Laborversuch verwendet wird 51

Abbildung 43: Die drei Reiter des Java Beispielprojektes 53
Abbildung 44: Die main-Methode in der Klasse "MongoGUI" 54
Abbildung 45: Beispiel für eine Aufgabenstellung für Laborteilnehmende..... 55
Abbildung 46: Ansicht des ersten Aufgabenbereiches 56
Abbildung 47: Eingliederung der JSON Struktur..... 57
Abbildung 48: Die zweite Aufgabenstellung mit dem Ergebnis der ersten Buttons 58
Abbildung 49: Beispielergebnis beim Einfügen eines Dokumentes 59
Abbildung 50: Ein User Dokument, nachdem das Passwort eingefügt wurde 60
Abbildung 51: Ein Post Dokument, das eingefügt wurde 61
Abbildung 52: Ein Kommentar, der zu einem bestimmten Post hinzugefügt wurde 62
Abbildung 53: Ein Post Dokument, welches auf ein User Dokument referenziert 63
Abbildung 54: Alle User, die ein größeres Alter als 25 haben 64
Abbildung 55: Ein Dokument, das beiden Anfragen entspricht 65
Abbildung 56: Die Kontaktfelder als Resultat der Aufgabe..... 66
Abbildung 57: Die Anzahl der Posts pro User 66
Abbildung 58: Ein Ausschnitt von zwei Dokumenten mit gleicher ID 67
Abbildung 59: Die Anzahl der Kommentare eines Users..... 68
Abbildung 60: Der unvollständige Social Media Post ohne Daten 69
Abbildung 61: Ein Post Dokument, welches auf ein Bild referenziert. 70
Abbildung 62: Die angeforderten Daten aus dem Post Dokument..... 70
Abbildung 63: Der Name des Users, der den Post geschrieben hat 71
Abbildung 64: Die ID des Bildes, das zum Post gehört 71
Abbildung 65: Alle Kommentare zum Post..... 72
Abbildung 66: Ein neuer Kommentar im Post..... 72
Abbildung 67: Ein vollständig ausgefüllter Post 73

12. Anhang

NoSQL Datenbank

Laborversuch

Thema:

MongoDB mit Java

Inhalt

1. Verbindung mit Robo3T 87
1.1. Grundlagen zu Interaktion mit der MongoDB 88
1.2. Kollektionen und Daten in Robo3T 89
1.3. Ein Dokument in Robo3T 89
1.4. Ein Key/Value Paar in Robo3T 90
1.5. Ein Key/Value Paar in JSON 90
1.6. Ein Key/Value Paar in Java 90
2. Einführung in das Social-Media Beispielprojekt 91
3. Interaktion mit dem Social Media GUI 92
4. Java Treiber Anwendung 93
4.1. Verbindungsaufbau 94
4.2. Eine Kollektion definieren 94
4.3. Erstellung von Dokumenten 94
4.4. Einen Array einfügen 95
4.5. Ein Embedded Document einfügen 95
4.6. Dokument in die Kollektion einfügen 95
4.7. Anfragen von Dokumenten 96
4.8. Filtern von Dokumenten 97
4.9. Aggregation Pipeline 98
5. Aufgaben 99
5.1. Verbindungsaufbau zur MongoDB 99
5.2. Aufgabe 1: JSON und BSON 100
5.3. Aufgabe 2: Dokumente mit Java erstellen 102
5.4. Aufgabe 2: Daten mit MongoDB Operatoren erfassen 105
5.5. Aufgabe 3: Adressierung eines Social-Media Posts 108

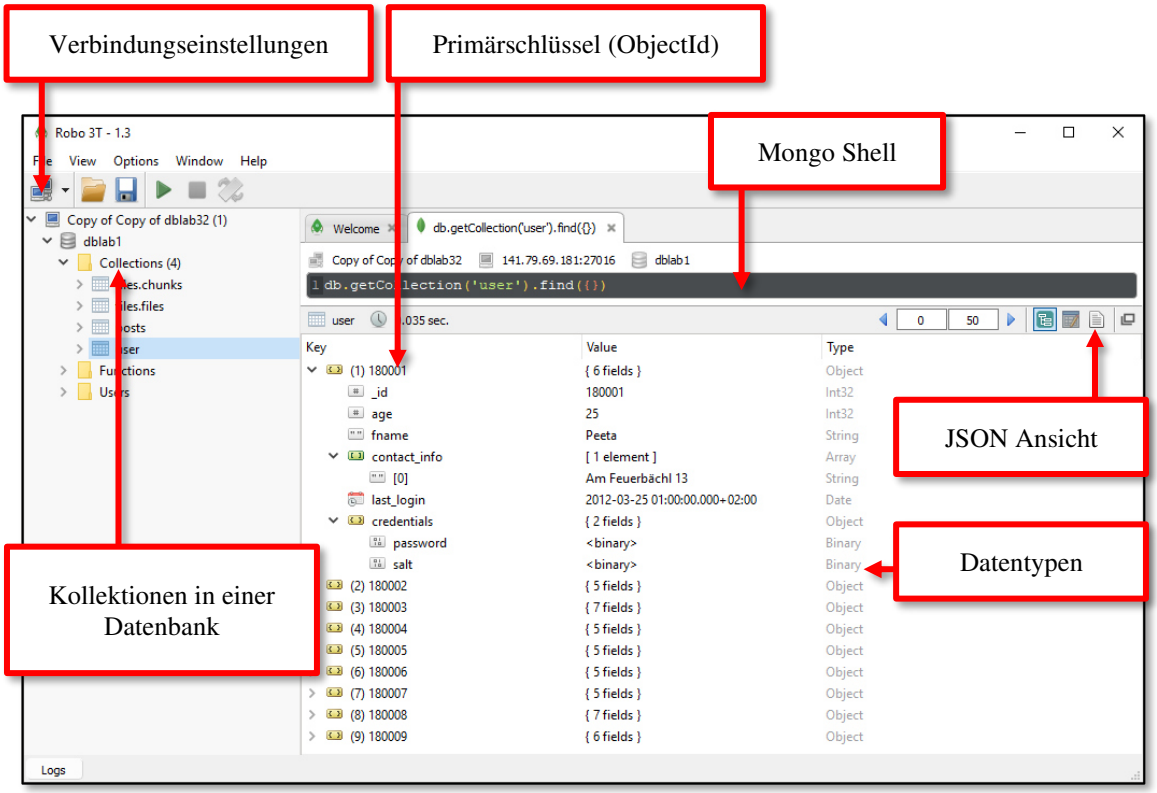
1. Verbindung mit Robo3T

Um direkt mit den Dokumenten innerhalb der MongoDB zu interagieren, gibt es verschiedene Möglichkeiten. Die einfachste wäre, direkt mit der Mongo Shell auf die Datenbank zuzugreifen. Die Interaktion mit einer Datenbank über eine Konsole, ist aber eher unpraktisch. Dies wird meist nur für administrative oder konfigurierende Arbeiten verwendet.

Einige Entwickler haben deshalb Tools entwickelt, mit denen es möglich ist, den Datenbestand der MongoDB gut zu überblicken.

Robo3T ist eine grafische Oberfläche, mit der sich MongoDB Datensätze sehr überschaubar betrachten lassen. Dieses GUI ist als freie Software erhältlich und kann in einer portablen Version ohne Installation ausgeführt werden. Es bietet die Möglichkeit an, die Dokumente im JSON-Format zu betrachten. Zugleich ist eine Mongo Shell integriert worden. Der volle Umfang der Mongo Shell API kann hierbei verwendet werden. Dies ist sehr praktisch, da auch Fehlermeldungen und Rückgabewerte zu sehen sind. Das erzeugt ein nahezu identisches Abbild zur direkten Interaktion mit der Mongo Shell.

Innerhalb des Laborversuches wird Roto3T verwendet, um Informationen aus dem Datenbestand zu erhalten. Datentypen, Embedded Documents und Primärschlüssel können hier eingesehen, entnommen oder auf ihre Korrektheit überprüft werden.

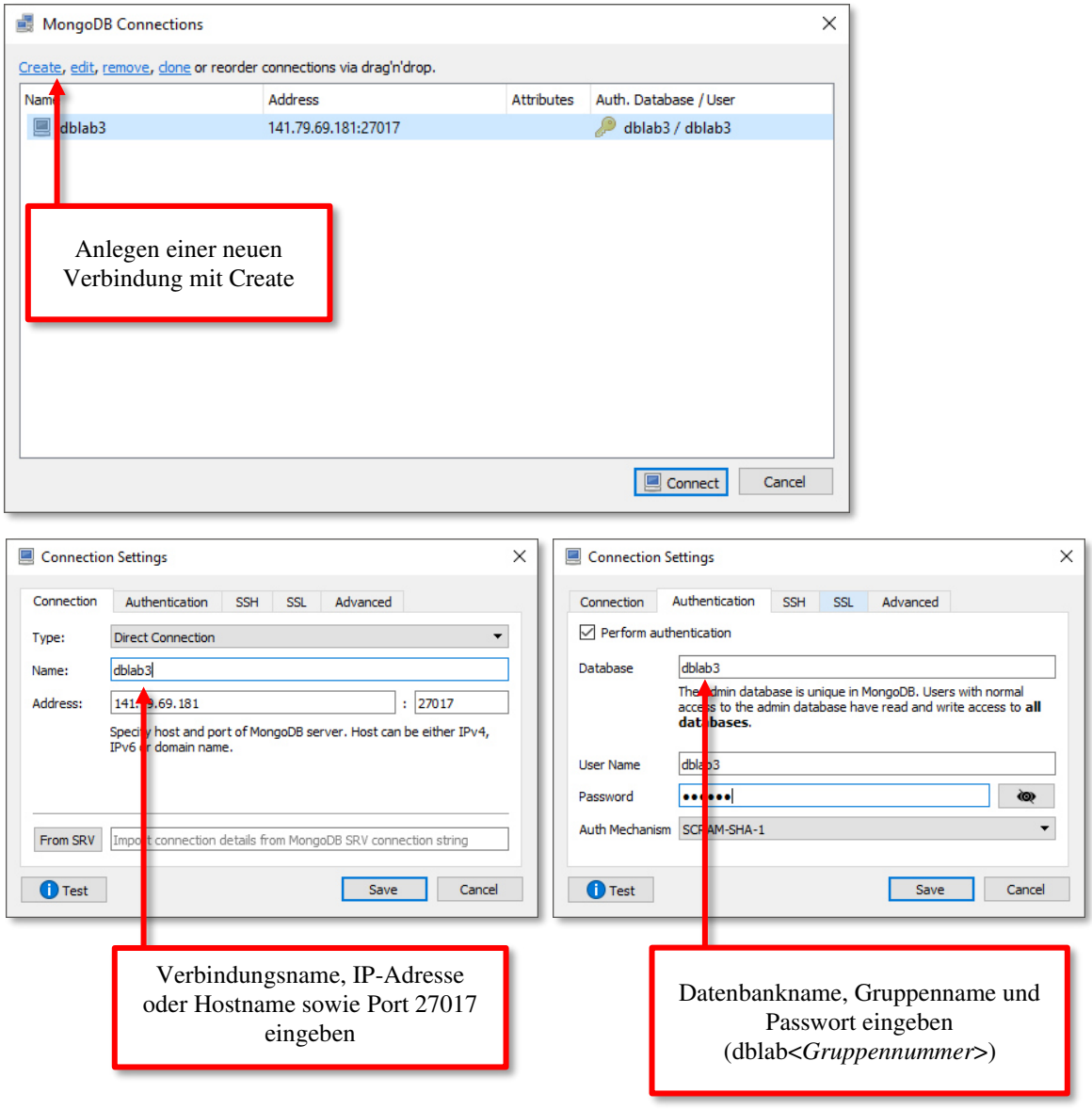


1.1. Grundlagen zu Interaktion mit der MongoDB

Innerhalb dieses Laborversuches sollen die wichtigsten Merkmale der dokumentenorientierten Datenbank MongoDB erlernt werden.

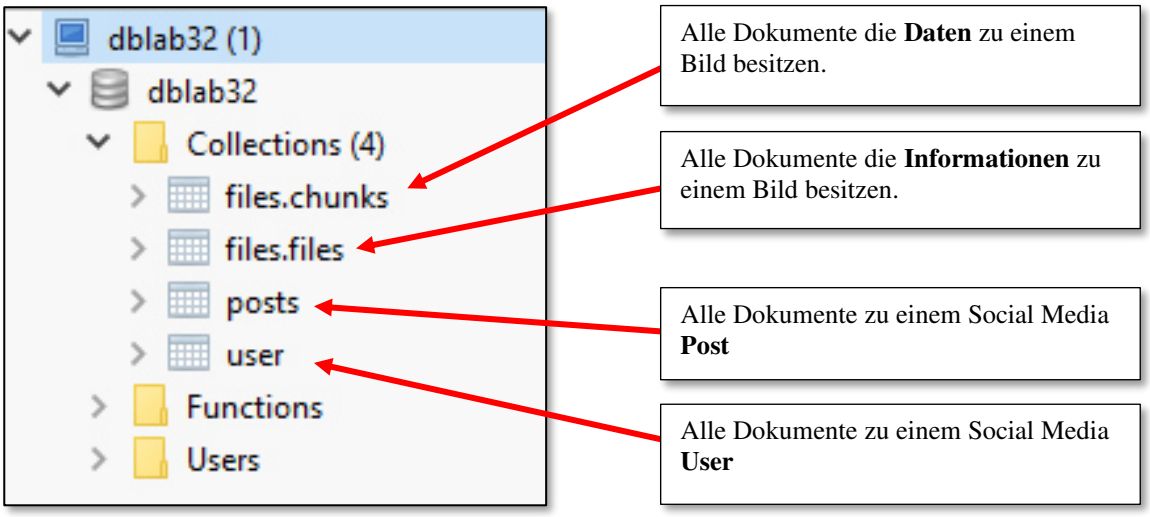
Dazu werden vier Tools angeboten, mit denen der Laborversuch durchgeführt wird.

1. Robo3T.
2. Die aktuellen MongoDB Java Treiber.
3. Eine vordefinierte und integrierte Datenbank. Dieses enthält bereits einige Datensätze.
4. Ein Beispielprojekt, welches erweitert bzw. vervollständigt werden soll.



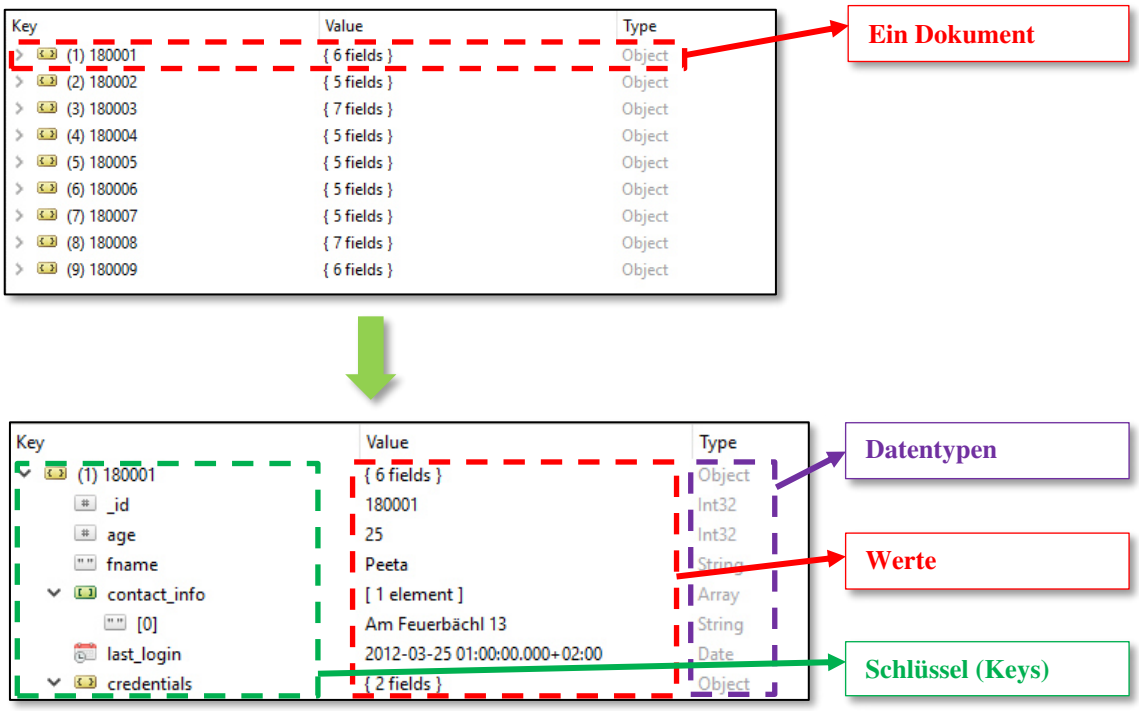
1.2. Kollektionen und Daten in Robo3T

Kollektionen: Enthalten Dokumente ähnlich wie eine Tabelle, aber ohne eine definierte Struktur. Eine Kollektion kann somit viele verschiedene Dokumente enthalten. Das Anlegen von verschiedenen Kollektionen ist aber trotzdem sinnvoll um die unterschiedlichen Inhalte separiert adressieren und organisieren zu können.



1.3. Ein Dokument in Robo3T

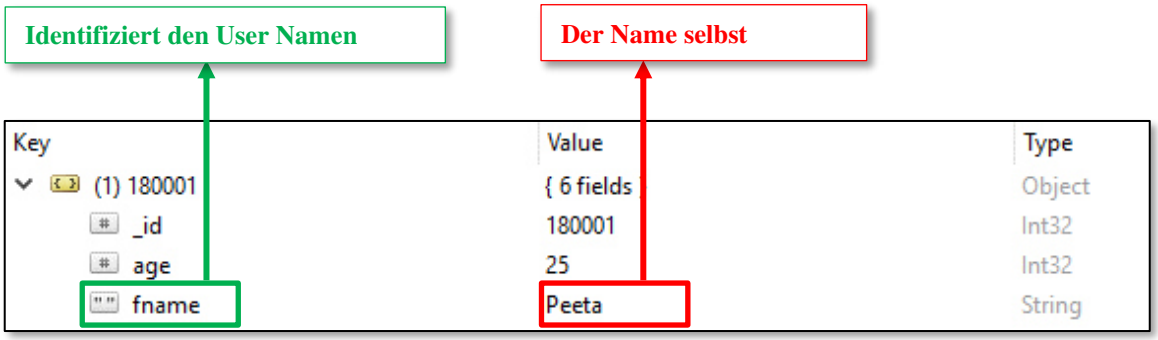
Alle Dokumente werden mit dem Primärschlüssel angezeigt. Wird das Dokument geöffnet, dann sind alle Werte und Datentypen zu sehen.



1.4. Ein Key/Value Paar in Robo3T

In MongoDB werden die einzelnen Datenfelder als Key/Value Paare dargestellt. Das bedeutet, dass ein Key (nicht zu verwechseln mit Primär- oder Fremdschlüssel) ein einzelnes Wertefeld identifizieren kann.

Beispiele für ein **Key/Value (Schlüssel/Wert)** Paar:



1.5. Ein Key/Value Paar in JSON

Alle User haben einen Schlüssel "fname", aber nicht alle User haben denselben Namen als "Wert".

```
{
    "fname" : "Peeta"
}
```

1.6. Ein Key/Value Paar in Java

Hier ist zu sehen, wie ein Dokument in Java dargestellt wird.

```
Document input = new Document();
input.append("fname","Peeta");
```

2. Einführung in das Social-Media Beispielprojekt

Eine grafische Benutzeroberfläche wurde implementiert, um die Adressierung einer vordefinierten Applikation zu simulieren. Das Ziel ist es, die einzelnen Elemente im GUI so zu adressieren, dass die implementierten Codezeilen die korrekten Resultate liefern.

Innerhalb des Laborversuches wird die Interaktion mit der Datenbank von den Laborteilnehmern nicht komplett programmiert, sondern lediglich **vervollständigt**. Das GUI dient lediglich zur Visualisierung und bietet, bis auf die Integration der Treiber und der Methoden für die Aufgabenstellungen, keine weitere Interaktion mit der MongoDB an.

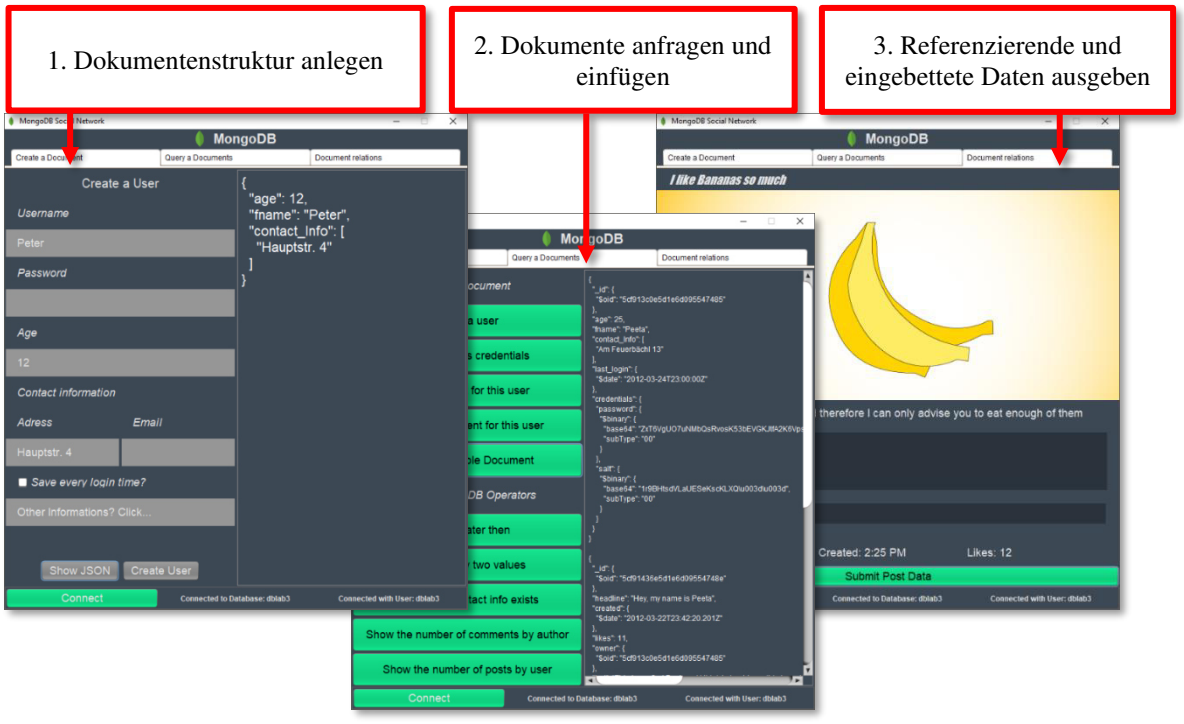
Um mit Daten aus einer MongoDB korrekt zu interagieren, müssen die Key/Value Paare aus den Dokumenten gefiltert werden. Von der MongoDB werden hierfür Operatoren zur Anfrage bereitgestellt. Diese Operatoren sind, in allen Treibern und auch in der Mongo Shell, zum Großteil gleich. Das Anwenden dieser Operatoren soll somit die Grundkenntnisse bei der Interaktion mit der MongoDB erläutern. Diese sollen innerhalb der Aufgabenstellungen verwendet werden, um die gewünschten Ergebnisse zu erzielen.

Das GUI ist in drei Tabs unterteilt, die jeweils einen Aufgabenbereich repräsentieren:

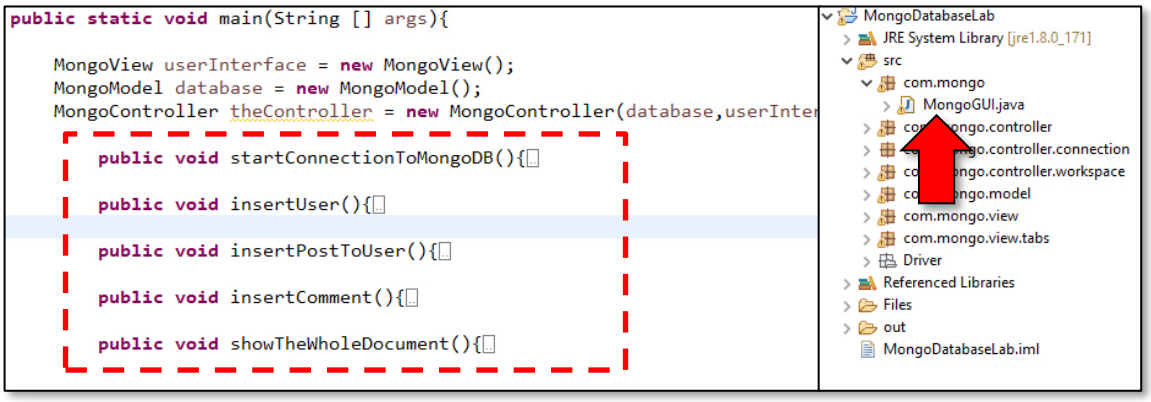
1. **Neue Dokumente erstellen**:
   Hierbei soll ein Formular verwendet werden, um ein Dokument zu erstellen und im JSON-Format anzuzeigen. Die einzelnen Bestandteile des Dokumentes sollen dabei erläutert werden. Ziel ist die Erkennung der Datentypen und Einteilung der Struktur durch die Laborteilnehmer.

2. **Dokumente aus der Datenbank anfragen und einfügen**:
   Hierbei sollen die Grundlagen zur Interaktion mit der MongoDB erläutert werden. Dazu zählen das Erstellen eines eigenen Dokumentes. Das Anfragen von Key/Value Paaren. Das Anfragen von referenzierenden und eingebetteten Dokumenten und die Verwendung der MongoDB Operatoren.

3. **Beziehungen zwischen Dokumenten verwenden**:
   Verschiedene Datentypen sollen aus unterschiedlichen Kollektionen und derer Dokumente entnommen werden. Hierbei sollen die Methoden so erweitert werden, dass alle Informationen aus einem Dokument korrekt wiedergegeben werden.

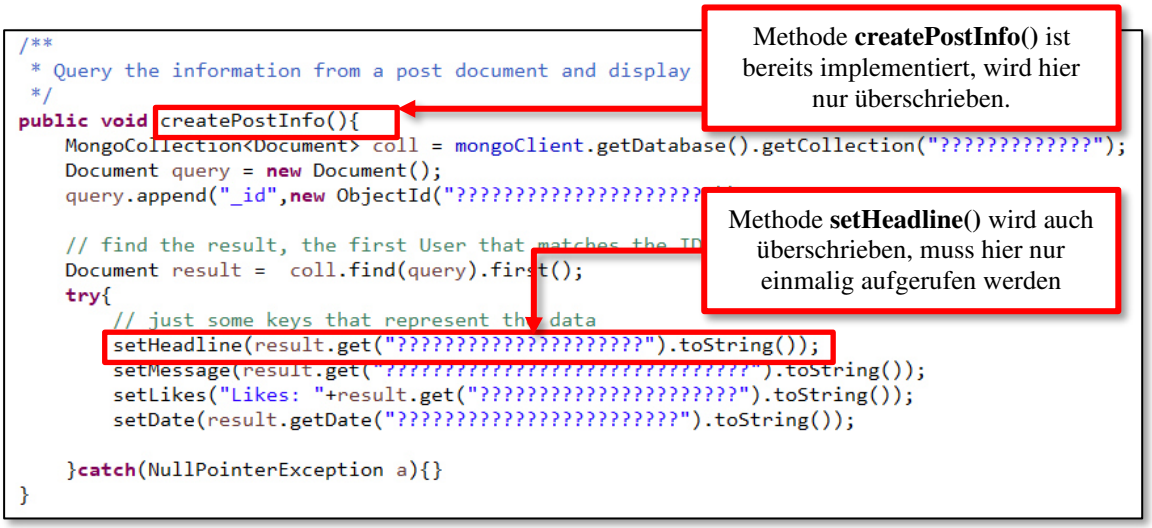


3. Interaktion mit dem Social Media GUI



Das Java Projekt enthält eine **Klasse** "**MongoGUI**". Diese muss geöffnet werden. **Weitere Klassen werden nicht benötigt** (siehe roter Pfeil).

Innerhalb dieser Klasse finden Sie alle Methoden, die benötigt werden, um das GUI zu adressieren. Bei Programmstart sind diese Methoden nur zum Teil ausgefüllt. Sie müssen von Ihnen selbst vervollständigt werden.



Im obigen Beispiel wird die Methode **createPostInfo()** bei einem Klick auf den ***"Submit Post Data"*** Button im dritten Tab ausgelöst. Durch die **setHeadline()** wird das Resultat an das GUI weitergegeben.

Jeder Button und jede Aufgabe besitzen solch eine Methode. Es muss nur der **Inhalt** der Methoden definiert und das **Resultat** ausgegeben werden. **Alle** Methoden sind in der Klasse MongoGUI bereits enthalten. Es müssen somit **keine** Methoden neu implementiert werden.

In der Aufgabenstellung sind detaillierte Informationen über alle GUI Buttons und deren Methoden zu finden.

4. Java Treiber Anwendung

Die Interaktion mit einer MongoDB findet durch die verschiedensten Applikationen statt. Dies können z.B. Apps, Webseiten, Online-Shops und Desktop-Programme sein. Die MongoDB bietet deshalb eine Vielzahl von Treibern an z.B. für PHP, C#, Python oder Java.

In diesem Laborversuch soll die MongoDB mit den offiziellen MongoDB Java Driver adressiert werden. Hierzu sind die notwendigen Java Archive (JAR) importiert worden. Die Dateien sind:

- **BSON Treiber**: bson-3.10.1.jar

  Enthält das BSON Package (org.bson), welches alle Klassen beinhaltet die notwendig sind, um Dokumente mit den BSON Datentypen anzulegen und auszulesen.

- **Kerntreiber**: mongodb-driver-core-3.10.1.jar

  Enthält das gesamte Package (com.mongodb) um mit der MongoDB zu interagieren.

- **Synchron Treiber**: mongodb-driver-sync-3.10.1.jar

  Enthält das Package (com.mongodb.client) welches die MongoClient Klasse beinhaltet. Hiermit können die statischen Methoden verwendet werden, um eine Verbindung mit der Datenbank herzustellen. Auch werden hier alle Operationen miteingebunden, die es ermöglichen, Anfragen und Filtermethoden an die Datenbank zu stellen.

Alle drei Treiber werden benötigt, um mit der MongoDB korrekt interagieren zu können.

4.1. Verbindungsaufbau

```
private MongoDatabase database; // Instanz der Datenbank
private MongoClient mongoClient; // Instanz einer MongoDB Verbindung
private MongoCredential credential; // Info zur Authentifizierung
```

- Diese Variablen repräsentieren die Verbindung, die Datenbank und die Authentifizierung.

```
ConnectionString connect = new ConnectionString(
   "mongodb://dblab3:dblab3@141.79.69.181:27017/?authSource=dblab3"
);
```

- Im *"ConnectionString"* sind der Benutzername, das Passwort und die Datenbank enthalten.

```
MongoClient mongoClient = MongoClients.create(connect);
database = mongoClient.getDatabase("dblab3");
```

- Anhand eines Strings in der Create Methode wird die Verbindung aufgebaut.
- Durch *"mongoClient.getDatabase()"* werden die Informationen zu der ausgewählten Datenbank gespeichert.

4.2. Eine Kollektion definieren

```
MongoCollection<Document> coll = database.getCollection("collectionName");
```

- *"getCollection()"* gibt die als String definierte Kollektion zurück.
- Die Instanz repräsentiert alle Dokumente aus dieser Kollektion.
- Alle weiteren Informationen zum Datenbestand werden dieser Instanz (also der Kollektion aus der MongoDB) entnommen.

4.3. Erstellung von Dokumenten

```
Document insertDoc = new Document();
```

- Ein einzelnes Dokument kann in Java mit der Klasse *"Document"* angelegt werden.

```
insertDoc.append("fname","Peter");
```

- Die Methode *"append()"* fügt ein Key/Value Paar in dieses Dokument ein.
- Jede *"append()"* Methode repräsentiert dabei genau ein Paar.

```
insertDoc.append("_id",new ObjectId("5cc45fc085c0759adf2f0664"));
```

- Values können Strings, Integer oder die verschiedenen BSON Datentypen sein .

4.4. Einen Array einfügen

```
Document insertDoc = new Document();
Document doc1 = new Document().append("Key","Value");
Document doc2 = new Document().append("Key","Value");

int integer = 5;
String string = "Something";
```

- Es können Dokumente oder Variablen in einem Array eingefügt werden.

```
insertDoc.append("ArrayKey",Arrays.asList(doc1, doc2));
insertDoc.append("ArrayKey2",Arrays.asList(integer, string));
```

- Ein Dokument, welches einen Array enthält, wird in die Datenbank eingefügt.
- Dabei wird dem Array Feld eine Liste von Objekten übergeben.

4.5. Ein Embedded Document einfügen

```
Document embeddDoc = new Document();
embeddDoc.append("key",value);
insertDoc.append("embeddDocKeyName", embeddDoc);
```

- Ein Embedded Document wird über einen einzelnen Key eingefügt.
- Unter diesem Key sind alle weiteren Key/Value Paare zu finden.

4.6. Dokument in die Kollektion einfügen

```
MongoCollection<Document> coll = database.getCollection("collectionName");
```

- Eine *"MongoCollection"* Instanz wird hierfür benötigt.

```
coll.insertOne(insertDoc);

List<Document> list = new ArrayList<Document>();
list.add(doc1);
list.add(doc2);

coll.insertMany(list);
```

- Methoden wie *"insertOne"* oder *"insertMany"* können verwendet werden, um in die Kollektion ein Dokument oder eine Liste von Dokumenten einzufügen.

4.7. Anfragen von Dokumenten

```
MongoCollection<Document> coll = database.getCollection("collectionName");
```

- Eine *"MongoCollection"* Instanz wird hierfür benötigt.

```
Document result = coll.find();  !!!
```

- Die Methode *"find()"* sucht nach allen Dokumenten.
- Gibt mehr als ein Objekt zurück, deshalb **Fehler** bei dieser Implementierung.

```
Document result = coll.find().first();
```

- Die Methode *"first()"* gibt nur das erste Objekt zurück.

```
MongoCursor<Document> cursor = userColl.find().iterator();
```

- Ein Iterator liefert bei jedem Fund ein Objekt zurück.
- Muss in einem *"MongoCursor"* angelegt werden.
- *"MongoCursor"* iteriert durch ein Ergebnis bei einer Anfrage, die mehr als ein Dokument enthält.
- Die Instanz *"Cursor"* repräsentiert nun alle gefundenen Objekte.

```
while (cursor.hasNext()) // wenn noch Objekte vorhanden sind
{
  System.out.println(cursor.next()); // gebe das nächste aus
}
```

- Alle Dokumente, die der Cursor findet, werden hier ausgegeben.

```
cursor.close();
```

- Der Cursor sollte nach dem Anfordern der Daten wieder geschlossen werden.

4.8. Filtern von Dokumenten

Vergleichsdokument erstellen

```
Document query = new Document();
query.append("fname","Edgar");

Document result = coll.find(query).first();
```

- Filtern mit Hilfe eines neu erstellten Dokumentes.
- Vergleicht Key/Value Paare vom Dokument mit der Kollektion.
- Zurückgegeben wird das passende Dokument.
- **Achtung**: Wird nach einem Nicht-Schlüsselattribut gefiltert, wird mit der Methode *"first()"* das erste Dokument, welches gefunden wird, zurückgegeben. Auch wenn mehrere Dokumente den Kriterien entsprechen.

Filter für Dokumente anlegen

```
Document result = coll.find(Filters.gt("age",50)).first();
```

- *"Filters.gt("age", 50)"* sucht nach Dokumenten die ein Feld *"age"* mit einem Wert beinhalten, der größer als 50 ist.
- Filters beinhaltet statische Methoden, welche die Abfrage nach verschiedenen Kriterien ermöglicht.

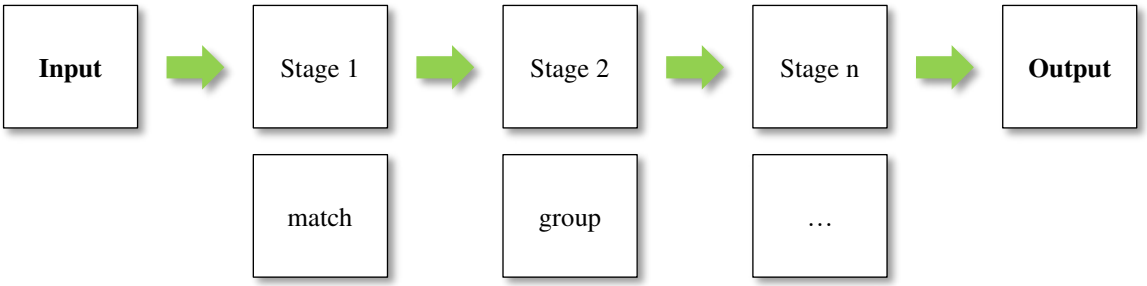
Weitere Filterkriterien sind:

- eq(fieldName, value) equals
- gte(fieldName, value) greater than or equal
- in(fieldName, values) if values is in e.g. a List
- lt(fieldName, value) less than
- lte(fieldName, value) less than or equal
- ne(fieldName, value) not equal
- regex(fieldName, pattern) regular expression
- text(String) match given search String
- size(fieldName, size) array size

4.9. Aggregation Pipeline

Eine der wichtigsten Methoden der MongoDB. Erlaubt es Dokumente anzufragen und diese durch mehrere Stationen (Stages) durchzuführen. Der Output einer Stage ist somit der Input der nächsten Stage. Hierbei können viele verschiedene Stages zum Einsatz kommen. Eine davon, mit der Sie in den Aufgaben konfrontiert werden, ist die *"group"* Stage.

```
AggregateIterable<Document> result = collection.aggregate(
 Arrays.asList(
    Aggregates.match(fields(exists("age", true))),
    Aggregates.group("$age", Accumulators.avg("age", 1))
 )
);
```



4.9.1. Stages

- **match**
  Findet Dokumente, die einem bestimmten Kriterium entsprechen. Alle Dokumente, die dabei gefunden wurden, werden an die nächste Stage übergeben.

- **group**
  Diese Stage zählt ein bestimmtes Feld und gibt dessen Anzahl wieder. Dabei kann durch die Klasse *"Accumulators"* festgelegt werden, wie gezählt wird. Beispiel *"sum()", "min()", "avg()"*.

Weitere (nicht im obigen Beispiel enthalten):

- **unwind**
  Muss auf ein Array Feld angewendet werden. Dieses wird dabei in einzelne Dokumente aufgeteilt und jedes Dokument an die nächste Stage weitergeleitet.

- **out**
  Das Ergebnis der gesamten Pipeline wird in einer neuen Kollektion ausgegeben. Muss deshalb immer als letztes implementiert werden.

4.9.2. Resultate der Pipeline

Das Ergebnis der Pipeline ist immer eine Vielzahl von Dokumenten. Deshalb muss zum Auslesen der einzelnen Inhalte eine *"for"* Schleife verwendet werden, um jedes einzelne Dokument wiederzugeben.

```
for(Document doc : result){
        System.out.println(doc);
}
```

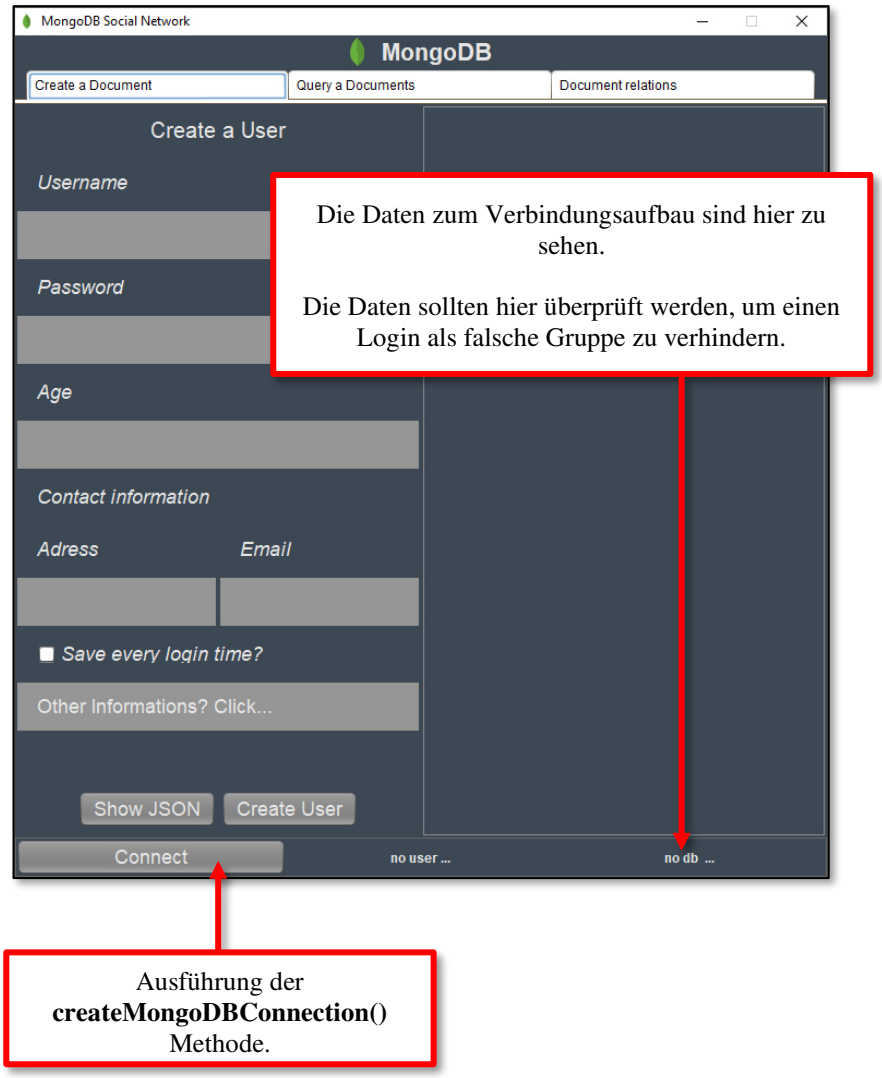
5. Aufgaben

5.1. Verbindungsaufbau zur MongoDB

Der erste Schritt sollte sein, eine Verbindung herzustellen. Innerhalb des Beispielprojektes ist eine unvollendete **createMongoDBConnection()** Methode implementiert worden.

```
public void createMongoDBConnection(){
    ConnectionString connectionString = newConnectionString(
   "mongodb://username?:password?@141.79.69.181:27017/?authSource=Database?"
    );
    MongoDatabase database = mongoClient.getDatabase();
  try {
    startConnectionToMongoDB(connectionString,"Database?");
  }catch (MongoClientException  mce){}
}
```

Um eine Verbindung herzustellen, sollte diese fertiggestellt werden. Wird das Projekt gestartet, kann mit dem Button *"Connect"* die Verbindung getestet werden. Bei einer erfolgreichen Verbindung kann damit begonnen werden, die erste Aufgabe zu bearbeiten.

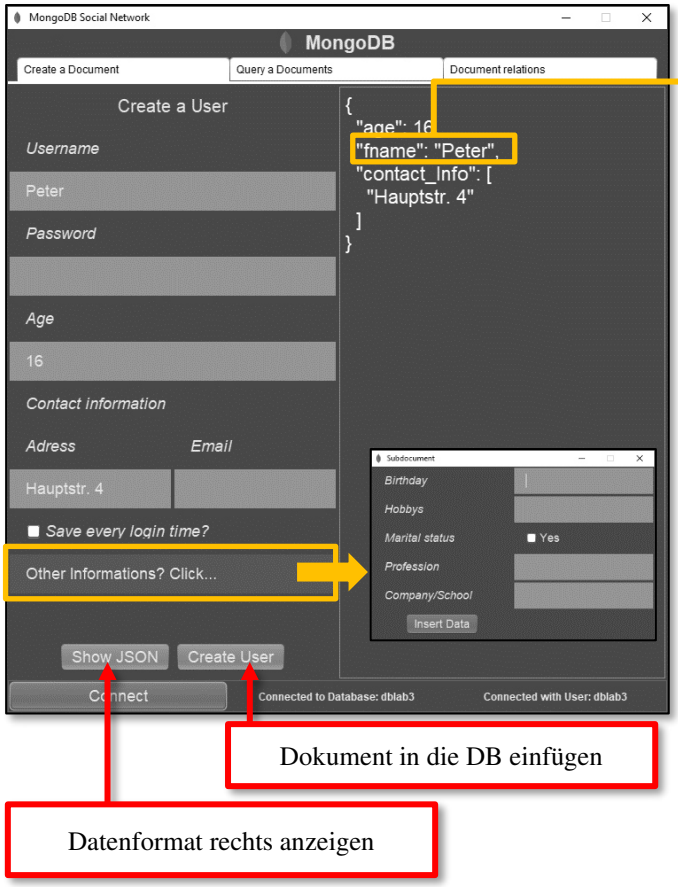


5.2. Aufgabe 1: JSON und BSON

5.2.1. Aufgabe 1: Erstellung von JSON Dokumenten

Die MongoDB ist eine dokumentenorientierte Datenbank. Um zu verdeutlichen, wie die Daten strukturiert sind und wie die einzelnen Datentypen angelegt werden, soll nun ein JSON Dokument erstellt werden. Verwenden Sie dazu das Formular im ersten Tab. Auf der **linken Seite** können die Informationen eingefügt werden. Auf der **rechten Seite** kann das JSON Datenformat betrachtet werden.

Fügen Sie nicht alle Informationen sofort ein, sondern schauen Sie sich nach und nach die einzelnen Datentypen an. Wenn Sie das Formular komplett ausgefüllt haben, dann sollten Sie das Dokument in die Datenbank einfügen und die Ergebnisse mit Robo3T vergleichen. Schreiben Sie die Datentypen in die Tabelle (nicht die Werte).



Schlüssel	Datentyp
Vorname	
Credentials	
Password	
Salz	
Alter	
Kontaktinformationen	
Adresse, Email	
Letzter Login	
Weitere Informationen	
Geburtstag	
Hobbys	
Familienstand	
Berufsstand	
Firma/Schule	

Was ist der Primärschlüssel des Dokumentes und wie heißt der Datentyp?
(Sehen Sie nach dem Einfügen in Robo3T nach)

Schlüssel	Datentyp

5.2.2. Aufgabe 1: Interpretation einer JSON Datenstruktur

Die MongoDB enthält mehr Datentypen als das JSON Format zulässt. Um dies zu gewährleisten wird das *"Extended JSON Format"* verwendet. Dieses speichert eine JSON Datei so ab, dass die erweiterten MongoDB Datentypen mit einem **$ Zeichen** versehen werden. So kann die MongoDB in einer JSON Datei die Datentypen erkennen und auslesen.

MongoDB Datenstruktur

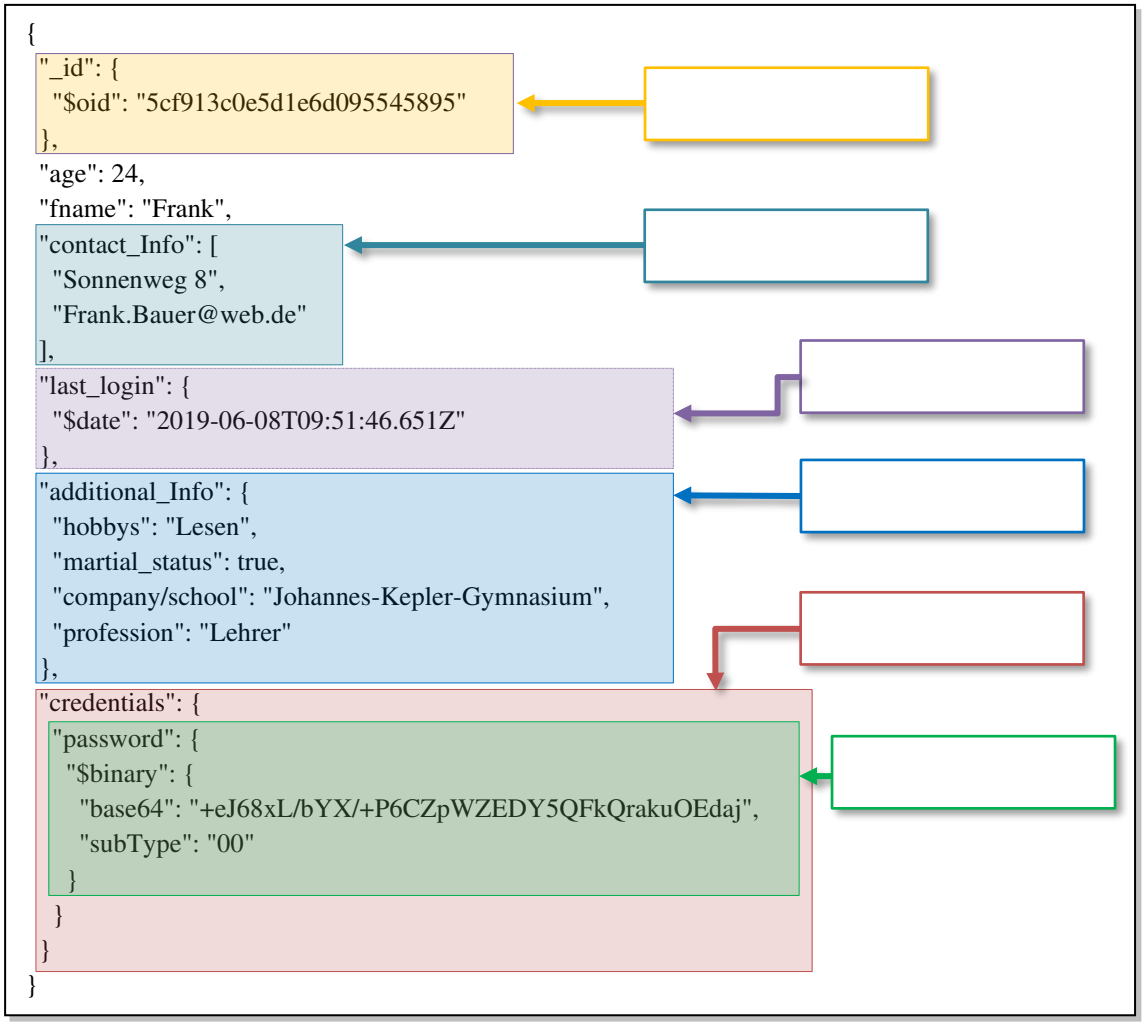
```
{
 "last_login" : ISODate("2019-06-14T14:32:39.973+02:00")
}
```

Extended JSON Datenstruktur

```
{
  "last_login": {
    "$date": "2019-06-14T12:31:41.807Z"
  }
}
```

Schreiben Sie in die freien Felder die passende Bezeichnung rein:

1) BSON Datentyp 2) Array 3) Embedded Document (Object)



5.3. Aufgabe 2: Adressierung der Daten mit Java Methoden

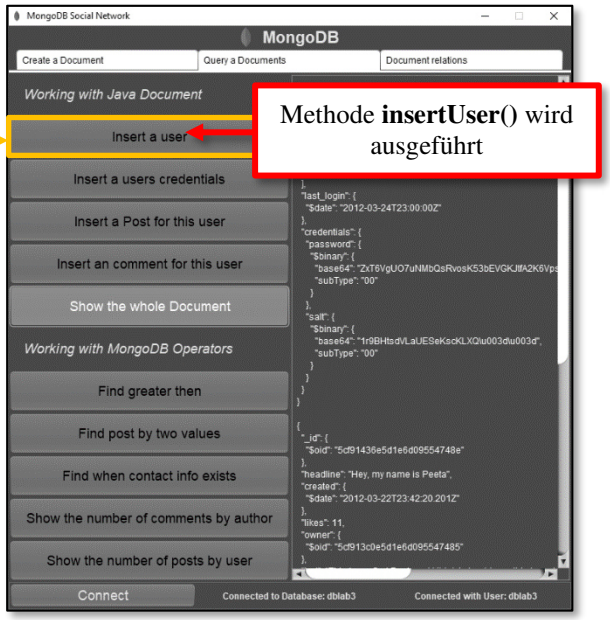
In dieser Aufgabe wird der zweite Tab des Beispielprojektes benötigt. Es soll wieder ein neues Dokument in die User Kollektion eingefügt werden. Hierbei soll dies aber mit Hilfe der Java Methoden durchgeführt werden.

Mit der Methode **printJsonResult()** können Sie die Ergebnisse im rechten Textfeld ausgeben, um sich die Dokumente als JSON Format anzeigen zu lassen.

5.3.1. Aufgabe 2: Neues Dokument erstellen und einfügen

Die Methode **insertUser()** soll erweitert werden. Definieren Sie hierfür die **richtige Kollektion**. Das Java Dokument soll einem User mit folgenden Schlüssel/Wert Paaren entsprechen:

- Primärschlüssel (ID)
- Alter
- Vorname
- Kontaktinformationen mit Email und Straße
- Dem letzten Login

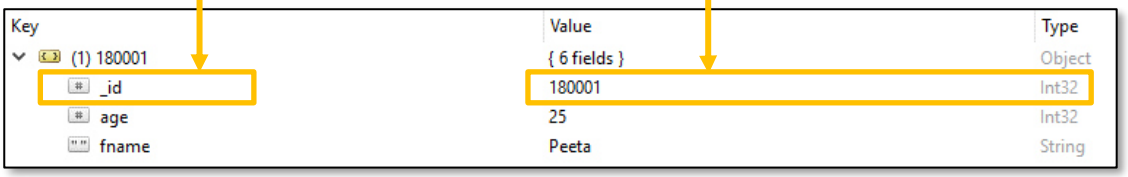



```
public void insertUser() {

        MongoCollection<Document> userCollection =
                            mongoClient.getDatabase().getCollection("???");

        Document myNewUser = new Document();
        myNewUser.append("_id", 0);
        myNewUser.append("age", 0);
        myNewUser.append("fname","????????????????");
        myNewUser.append("contact_info",Arrays.asList("????????????????",
                                                      "????????????????
                                                    ?"));
        myNewUser.append("last_login",new Date());

        userCollection.insertOne(myNewUser);
        printJsonResult(myNewUser);
}
```



5.3.2. Aufgabe 2: Passwort für einen User generieren

Die Methode **generateAndInsertPassword()** soll ein Dokument mit dem Passwort als String und dem Salz als Datentyp Byte erstellen. Diese Informationen sollen zum User Dokument aus der vorherigen Aufgabe hinzufügt werden.

Füllen Sie die Methode so aus, dass:

- Ein Passwort definiert wurde (Denken Sie sich ein beliebiges aus).
- Beide Schlüssel im Dokument *"credentials"* enthalten sind.
- Die updateOne() Methode das Dokument mit der richtigen ID updatet.

Die updateOne() Methode erhält zwei Parameter, welche bereits eingetragen wurden:

1. Welches Dokument soll aktualisiert werden? (eq = equals)
2. Wo soll der Array im Dokument eingefügt werden? (set(key,value) = neues Feld)

5.3.3. Aufgabe 2: Ein User schreibt einen Post

Die Methode **insertPostWrittenByUser()** erstellt einen neuen Post in der Post Kollektion. Dies soll ebenfalls wie das User Dokument vervollständigt werden.

Füllen Sie die Methode so aus, dass der Benutzer mit dem Namen **Susanna** einen neuen Post mit folgenden Inhalten geschrieben hat:

- Die Überschrift, welche Sie frei wählen, enthalten ist.
- Der Post eine beliebige Anzahl an Likes besitzt.
- Die ID des Besitzers (owner) korrekt eingetragen ist.
- Der Post einen von Ihnen erfundenen Text beinhaltet.

5.3.4. Aufgabe 2: Der Post wird kommentiert

Es soll zu dem ersten Post Dokument, das auf den User mit dem Namen Susanna referenziert, ein Kommentar eingefügt werden. Dies geschieht mit Hilfe der Methode **insertComment()**. Vervollständigen Sie auch diese Methode. Um das *"Embedded Document"* in ein Post Dokument einzufügen, wird wieder eine Aktualisierung (updateOne Methode) verwendet.

Füllen Sie die Methode so aus, dass:

- Zum **ersten Post** den Susanna geschrieben hat, ein Kommentar eingefügt wird.
- Hierzu müssen Sie die ID (ObjectID) des Posts in der Post Kollektion finden.
- Der Kommentar dieselbe Struktur wie die anderen Kommentare im Datenbestand hat.
- Der Kommentar als neues Feld im Array angelegt wird.

Die updateOne() Methode erhält wieder dieselben zwei Parameter:

1. Welches Dokument soll aktualisiert werden? (eq = equals)
2. Wo soll der Array im Dokument eingefügt werden? (set(key,value) = neues Feld)

5.3.5. Aufgabe 2: Anzeigen der User und Post Dokumente von Susanna

Zuletzt sollen alle Daten zu Susanna ausgegeben werden. Vervollständigen Sie die **showAllInsertedDocuments()** Methode, um das User Dokument und den darauf referenzierenden Post von Susanna, anzuzeigen.

Dazu müssen Sie in der Methode die:

- Zwei Kollektionen festlegen.
- Die ID des Users in einem Vergleichsdokument (queryUser) einfügen.
- Die ID des Users soll auch verwendet werden, um den richtigen Post zu finden. Lesen Sie den Schlüssel aus dem Dokument wieder aus (queryUser.get("Schlüssel")).
- Übergeben Sie in beide find() Methoden das jeweilige Vergleichsdokument.

Sollten Sie alles ausgefüllt haben, dann testen Sie die Methoden im GUI.

Welche Beziehungen und welche Kardinalitäten liegen vor?

Beziehungen: "*Document References*" oder *"Embedded Document"*?

Kardinalitäten: 1:1 oder 1:N?

- **Welche Beziehung besteht zwischen dem User und den Credentials?**

  Antwort

  Bezeichnung: _____

  Kardinalität: _____

- **Welche Beziehung besteht zwischen dem Post und dem Kommentar?**

  Antwort

  Bezeichnung: _____

  Kardinalität: _____

- **Welche Beziehung besteht zwischen dem User und dem Post?**

  Antwort

  Bezeichnung: _____

  Kardinalität: _____

5.4. Aufgabe 2: Daten mit MongoDB Operatoren erfassen

In diesem Aufgabenbereich sollen die Grundlagen zur Interaktion mit den MongoDB Operatoren verdeutlicht werden. Dazu wird für jede Aufgabe ein eigener Button adressiert, der die Methode ausführt und an die Datenbank senden.

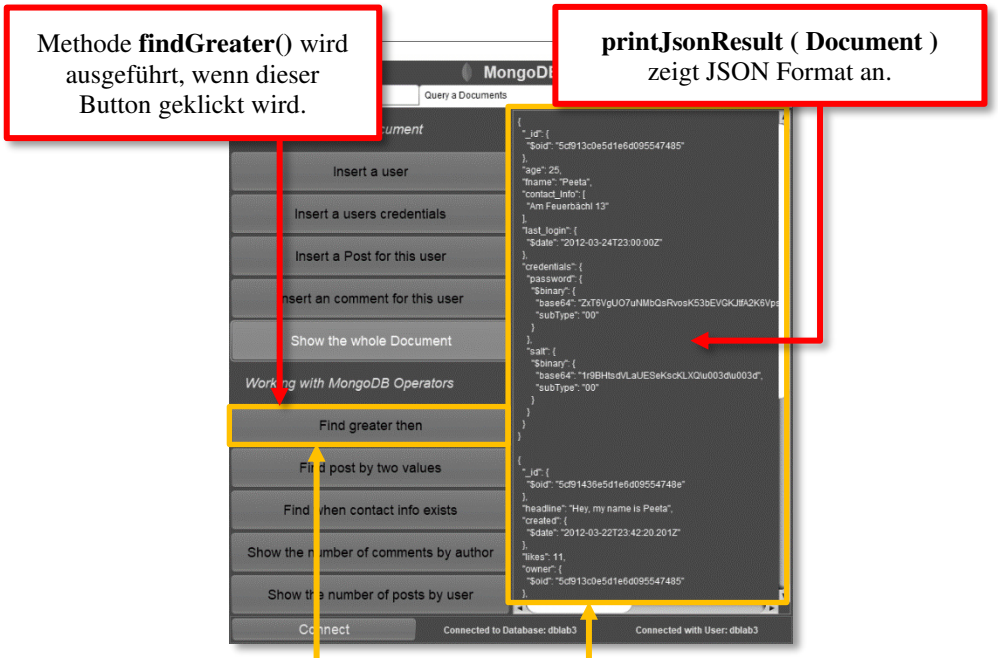
5.4.1. Aufgabe 2: Alle User nach Alter filtern

Der Größer als (Filters.gt) Operand wird verwenden, um alle User Dokumente auszugeben, die ein Alter größer als 25 haben.

Vervollständigen Sie die folgenden Angaben im Code:

- Den Schlüssel, der das Alter der User identifiziert.
- Das Alter 25 als Wert definieren.
- Nur den Namen des Users anzeigen lassen (Projections.include("????????")).

Beispiel:




```
public void findGreater(){
MongoCollection<Document> userCollection =
            mongoClient.getDatabase().getCollection("???????????");

MongoCursor<Document> userCursor =
            userCollection.find(
                  Filters.gt("???????",0)).projection(
                        Projections.include("????????")
                  ).iterator();

      while(userCursor.hasNext()){
            printJsonResult(userCursor.next());
      }
      userCursor.close();
}
```

5.4.2. Aufgabe 2: Verknüpfung von zwei Anfragen

Es soll ein UND Operator (Filters.and) verwendet werden, um die Posts zu finden, die noch **keine Kommentare** haben und wo das **Wort *"day"* in der Überschrift** vorkommt.

Dazu müssen Sie:

- Die Kollektion Posts definieren.
- Die Regular Expression (regex) mit dem Suchwort ausfüllen (Pattern.compile("????")).
- Den Schlüssel für das Feld eintragen, das nicht vorhanden sein soll (Filters.exists("??", false)).
- Den Schlüssel für das Feld eintragen, auf dem die Regular Expression angewendet werden soll (Filters.regex("??", regex)).

5.4.3. Aufgabe 2: Inhalte überprüfen

Zeigen Sie nur die Dokumente an, die eine **Kontaktinformation** besitzen. Verwendung von dem *"Filters.exists"* Operator.

Dazu müssen Sie:

- Den Schlüssel angeben, nach dem der Operator Filters.exists("??", true) filtern soll.

5.4.4. Aufgabe 2: Die Anzahl der Posts pro User

Geben Sie an, wie viele Posts ein Benutzer geschrieben hat. Dazu müssen Sie die Aggregation Pipeline vervollständigen. Diese beinhaltet eine Stage:

- **1 Stage**: Um die Posts den Usern zuzuordnen, wird der Schlüssel *"Owner"* gezählt. Geben Sie den Schlüssel für den Besitzer an.

```
Aggregates.group("$???", Accumulators.sum("A new field name", 1))
```

Hinweis: Das $ Zeichen muss bei Group immer angegeben werden, da dieser Operator einen *"Field Path"* als Parameter erhält. Das heißt, dass der String "$Schlüsselwert" den Weg zum Wert zurückgibt.

5.4.5. Aufgabe 2: Alle Kommentare eines User finden

Geben Sie an, wie viele Kommentare ein Benutzer geschrieben hat. Dazu müssen Sie die Aggregation Pipeline vervollständigen. Diese beinhaltet 4 Stages:

- **1 Stage**: Geben Sie den Schlüssel für die Kommentare an, um sicherzustellen, dass diese auch vorhanden sind

  Aggregates.match("???", true)

- **2 Stage**: Die Kommentare sind in einem Array. Geben Sie deshalb wieder den Schlüssel für die Kommentare an, um aus einem Array viele einzelne Dokumente zu erstellen.

  Aggregates.unwind("$???????")

- **3 Stage**: Eine Gruppierung zählt einen Schlüssel und gibt die Anzahl der gleichen Schlüssel zurück. Geben Sie den Schlüssel an, der auf einen User referenziert.

  Aggregates.group("$???", Accumulators.sum("A new field name", 1))

- **4 Stage**: Gibt das Ergebnis in einer neu erstellten Kollektion aus. Sie können hierbei einen beliebigen Namen vergeben.

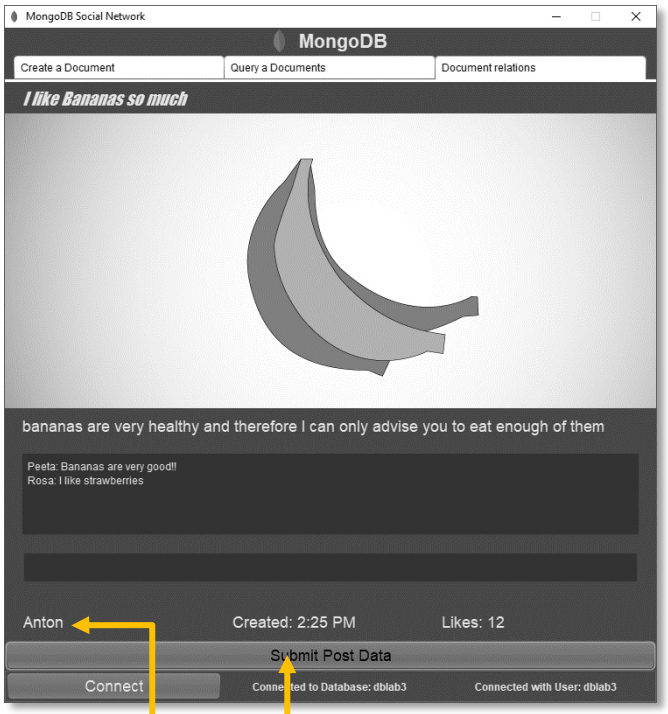
  Aggregates.out("A collection Name")

Hinweis: Vergessen Sie bei Group und Unwind nicht das $ Zeichen!

5.5. Aufgabe 3: Adressierung eines Social-Media Posts

Die Aufgabe hier ist es, ein vorgegebenes Design so zu adressieren, dass alle notwendigen Informationen enthalten sind. Im Gegensatz zur Aufgabe Zwei, sollen nicht einzelne JSON Dokumente ausgegeben werden, sondern nur die tatsächlich benötigten Informationen.

Ziel dieser Aufgabe ist es, dass ein Post korrekt angezeigt wird. Dazu müssen die Beziehungen der Dokumente verwendet werden, um die einzelnen Daten auslesen zu können.



Dabei werden die Dokumente, Embedded Documents und referenced Documents so adressiert, dass keine Klammern, IDs oder Datentypen angezeigt werden.

Bei Klick auf den Button *"Submit Post Data"* werden alle Methoden für den Post ausgeführt. Die Methode dieses Buttons ist bereits implementiert. Hierbei müssen Sie selbst keine Schritte unternehmen.

Für jedes Element im Post gibt es eine *"setter"* Methode. Diese benötigt zum Anzeigen die entsprechenden Daten aus der MongoDB.

```
public void createPostName(){

        // Code in Ecliplse...

        try{
          // Code in Ecliplse...
          setFirstName(userResult.get("???????").toString());

        }catch (Exception a){}
}
```

5.5.1. Aufgabe 3: Anfordern von Daten aus dem Post Dokument

Zu Beginn müssen Sie sich aus der Kollektion Posts einen Post heraussuchen.

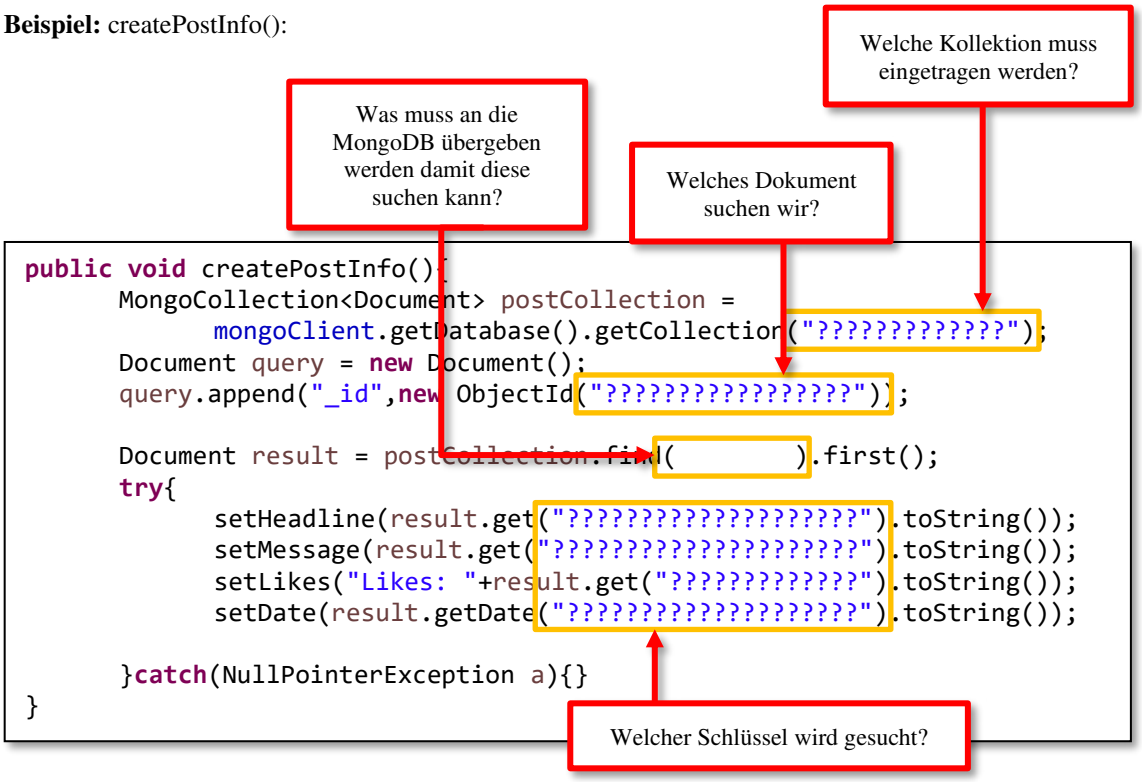
- Die Bedingung hierbei ist, dass es sich um einen Post mit Bild handelt.
- Wählen Sie einen Post mit Robo3T aus, der den Schlüssel *"Image"* besitzt.

Danach sollen alle Daten aus dem Post Dokument selbst angefordert werden. Dazu müssen Sie die Methode **createPostInfo()** vervollständigen.

Ihre Aufgabe besteht darin, dass:

- Die benötigte Kollektion zu definieren.
- Ein Vergleichsdokument mit der ID des Posts erstellen.
- Die *"find()"* Methode anhand des Vergleichsdokumentes den Post suchen lassen.
- Die Überschrift eines Posts anzeigen. (setHeadline())
- Die Post Nachricht. (setMessage())
- Die Likes anzeigen. (setLikes())
- Die Uhrzeit des Posts. (setDate())

Beispiel: createPostInfo():



Wenn sie das Projekt starten und im PostTab auf

"Submit Post Data"

klicken, dann werden die Anfragen an die Datenbank gesendet und die Ergebnisse angezeigt.

5.5.2. Aufgabe 3: Anfordern des User Namens

Die Methode **createPostName()** soll von dem User, der den Post geschrieben hat, den Namen ausgeben. Ihre Aufgabe besteht darin, dass:

- Beide Kollektionen definiert werden, die hierfür notwendig sind.
- Ein Vergleichsdokument erstellt wird, welches die ID des Posts enthält.
- Eine *"find()"* Methode anhand des Vergleichsdokumentes in der Post Kollektion den richtigen Post findet.
- In der User Kollektion anhand des angefragten Posts nach dem User mit der richtigen ID gesucht wird.
- Der Username aus dem gefundenen Dokument ausgegeben wird (userResult.get(?)).

5.5.3. Aufgabe 3: Integration von Mediendaten

Die Methode **createPostImage()** soll ein Image aus der Datenbank anzeigen. Ein Post zeigt durch einen Schlüssel auf ein bestimmtes Image. Ein Image zeigt aber auch auf einen Post. Entscheiden Sie selbst, wie Sie die ID auswählen.

5.5.4. Aufgabe 3: Anfordern von eingebetteten Kommentaren

Die Methode **createPostComments()** soll mehrere *"Embedded Documents"* ausgegeben. Hierbei wird eine Aggregation Pipeline verwendet. Vervollständigen Sie diese:

- **1 Stage**: Geben Sie den Schlüssel für den Post an, von dem Sie die Kommentare auslesen möchten.

  Aggregates.match(fields(eq("_id", new ObjectID(????))))

- **2 Stage**: Projections.computed() erstellt ein neues Dokument und erfasst dabei nur die Schlüssel/Wert Paare, die angegeben werden. In diesem Fall ist das der Array, der die Kommentare enthält. Geben Sie den Schlüssel zum gesuchten Array an, um alle Kommentare auszulesen.

  Projections.computed("new Document name","$??????")

Hinweis: Die Stage *"computed"* benötigt wieder das $ Zeichen!

5.5.5. Aufgabe 3: Einen neuen Kommentar einfügen

Um neue Kommentare in einem Post einfügen zu können, sollen Sie die Methode **insertAnewComment()** vervollständigen. Wenden Sie das bisher erlernte Wissen aus den vorherigen Aufgaben an, um dies zu ermöglichen.